ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN BIG DATA: TECNOLOGÍA Y ANALÍTICA AVANZADA

SISTEMA BASADO EN AGENTES PARA LA GENERACIÓN DE DATOS SINTÉTICOS A PARTIR DE DATA CONTRACTS

Autor: Álvaro Pérez Romero

Director: Mariana Teresa Olavarri Niño

Madrid

Junio 2026

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título “Sistema basado en agentes para la generación de datos sintéticos a partir de Data Contracts” en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2018/19 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos.

El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido

tomada de otros documentos está debidamente referenciada.

Fdo.: Álvaro Pérez Romero Fecha: 1/05/2026

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Mariana Teresa Olavarri Niño Fecha: 15/05/2026

Vº Bº del Coordinador de Proyectos

Fdo.: Carlos Morrás Ruiz-Falcó Fecha: ……/ ……/ ……

AUTORIZACIÓN PARA LA DIGITALIZACIÓN, DEPÓSITO Y DIVULGACIÓN EN RED DE PROYECTOS FIN DE GRADO, FIN DE MÁSTER, TESINAS O MEMORIAS DE BACHILLERATO

1º. Declaración de la autoría y acreditación de la misma.

El autor D. Álvaro Pérez Romero DECLARA ser el titular de los derechos de propiedad intelectual de la obra: Sistema basado en agentes para la generación de datos sintéticos a partir de Data Contracts, que ésta es una obra original, y que ostenta la condición de autor en el sentido que otorga la Ley de Propiedad Intelectual.

2º. Objeto y fines de la cesión.

Con el fin de dar la máxima difusión a la obra citada a través del Repositorio institucional de la Universidad, el autor **CEDE** a la Universidad Pontificia Comillas, de forma gratuita y no exclusiva, por el máximo plazo legal y con ámbito universal, los derechos de digitalización, de archivo, de reproducción, de distribución y de comunicación pública, incluido el derecho de puesta a disposición electrónica, tal y como se describen en la Ley de Propiedad Intelectual. El derecho de transformación se cede a los únicos efectos de lo dispuesto en la letra a) del apartado siguiente.

3º. Condiciones de la cesión y acceso

Sin perjuicio de la titularidad de la obra, que sigue correspondiendo a su autor, la cesión de derechos contemplada en esta licencia habilita para:

a) Transformarla con el fin de adaptarla a cualquier tecnología que permita incorporarla a internet y hacerla accesible; incorporar metadatos para realizar el registro de la obra e incorporar "marcas de agua" o cualquier otro sistema de seguridad o de protección.
b) Reproducirla en un soporte digital para su incorporación a una base de datos electrónica, incluyendo el derecho de reproducir y almacenar la obra en servidores, a los efectos de garantizar su seguridad, conservación y preservar el formato.
c) Comunicarla, por defecto, a través de un archivo institucional abierto, accesible de modo libre y gratuito a través de internet.
d) Cualquier otra forma de acceso (restringido, embargado, cerrado) deberá solicitarse expresamente y obedecer a causas justificadas.
e) Asignar por defecto a estos trabajos una licencia Creative Commons.
f) Asignar por defecto a estos trabajos un HANDLE (URL *persistente)*.

4º. Derechos del autor.

El autor, en tanto que titular de una obra tiene derecho a:

a) Que la Universidad identifique claramente su nombre como autor de la misma
b) Comunicar y dar publicidad a la obra en la versión que ceda y en otras posteriores a través de cualquier medio.
c) Solicitar la retirada de la obra del repositorio por causa justificada.
d) Recibir notificación fehaciente de cualquier reclamación que puedan formular terceras personas en relación con la obra y, en particular, de reclamaciones relativas a los derechos de propiedad intelectual sobre ella.

5º. Deberes del autor.

- El autor se compromete a:

a) Garantizar que el compromiso que adquiere mediante el presente escrito no infringe ningún derecho de terceros, ya sean de propiedad industrial, intelectual o cualquier otro.
b) Garantizar que el contenido de las obras no atenta contra los derechos al honor, a la intimidad y a la imagen de terceros.
c) Asumir toda reclamación o responsabilidad, incluyendo las indemnizaciones por daños, que pudieran ejercitarse contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.
d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción

de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a ……1….. de ……Mayo……... de …2026…….

ACEPTA

Fdo……

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

MÁSTER EN BIG DATA: TECNOLOGÍA Y ANALÍTICA AVANZADA

SISTEMA BASADO EN AGENTES PARA LA GENERACIÓN DE DATOS SINTÉTICOS A PARTIR DE DATA CONTRACTS

Autor: Álvaro Pérez Romero

Director: Mariana Teresa Olavarri Niño

Madrid

Junio 2026

SISTEMA BASADO EN AGENTES PARA LA GENERACIÓN DE DATOS SINTÉTICOS A PARTIR DE DATA CONTRACTS

Autor: Pérez Romero, Álvaro.
Director: Olavarri Niño, Mariana Teresa.
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

Se ha diseñado e implementado un sistema basado en cinco agentes LLM que genera datos sintéticos a partir de contratos formales en formato Open Data Contract Standard, combinando un generador determinista que garantiza la validez estructural con agentes de Azure OpenAI que aportan plausibilidad semántica. El sistema incorpora un bucle iterativo de auto-corrección con rollback y observabilidad completa OpenTelemetry, y se ha evaluado sobre una batería de diez contratos alcanzando convergencia estructural sin violaciones de claves foráneas en todos ellos.

Palabras clave: datos sintéticos, agentes LLM, data contracts, ODCS, OpenTelemetry, Microsoft Agent Framework

1. Introducción

Los datos sintéticos se han consolidado como respuesta a las restricciones regulatorias (GDPR, LGPD, CCPA) [13] que limitan el uso de datos reales en entornos no productivos. Las soluciones existentes se polarizan entre herramientas basadas en reglas (Faker, Mockaroo), que ignoran la estructura relacional, y modelos generativos (CTGAN, SDV, Mostly AI), que requieren datos reales como entrenamiento, son opacos y no integran contratos formales.

Este trabajo aborda esa brecha mediante un sistema que combina rigor estructural —un generador determinista que respeta el contrato por construcción— con flexibilidad semántica aportada por agentes LLM, bajo el principio rector "el LLM solo decide lo que no puede decidirse determinísticamente". Esta inversión reduce el coste por ejecución en aproximadamente un orden de magnitud y elimina por construcción las violaciones estructurales típicas de las soluciones puramente generativas.

2. Definición del proyecto

El sistema toma como entrada un contrato formal en formato Open Data Contract Standard (ODCS) v3.x [1] y produce como salida un dataset CSV multi-tabla validado contra el contrato, junto con un artefacto autocontenido de observabilidad: trazas OpenTelemetry [9], manifiesto JSON y un visor HTML standalone para análisis post-mortem.

Los seis objetivos específicos del trabajo —parser robusto de ODCS, generador determinista con soporte de PKs y FKs compuestas, ciclos y dominios saturables; capa de análisis multi-dimensional; bucle iterativo de auto-corrección con rollback;

observabilidad estándar OpenTelemetry; y visor HTML autocontenido— se han alcanzado en su totalidad.

3. Descripción del sistema

El sistema se organiza en cinco capas funcionales coordinadas por el Microsoft Agent Framework: entrada (parser ODCS), planificación, generación determinista, análisis multi-dimensional y diagnóstico-enriquecimiento iterativo. Una capa transversal de observabilidad atraviesa todas las anteriores.

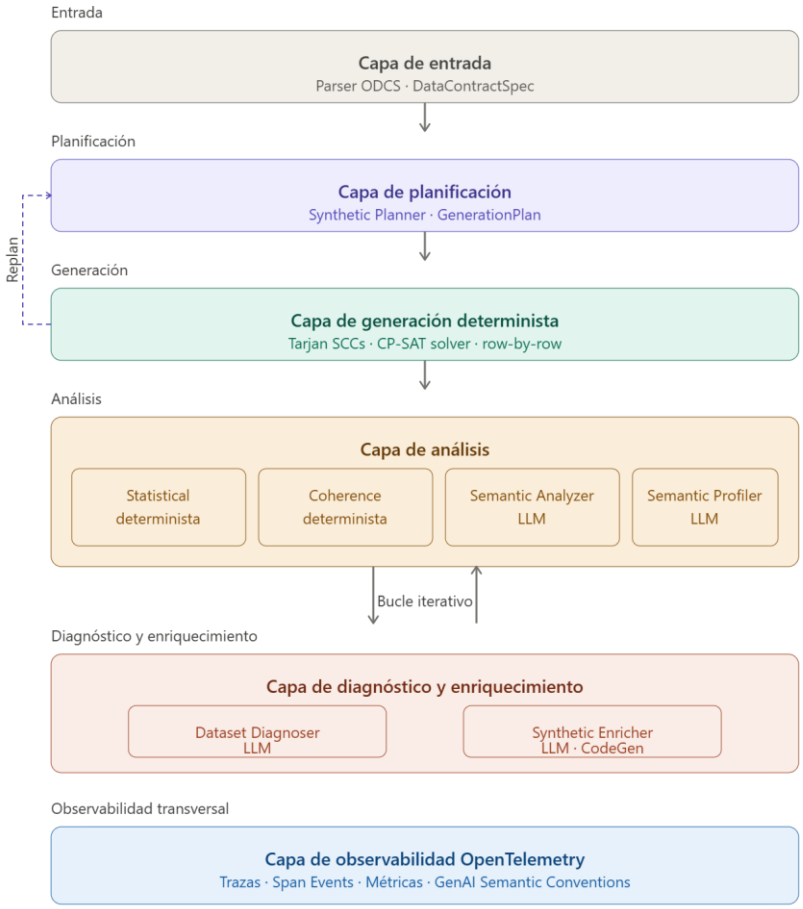

Ilustración 1 — Arquitectura del sistema en cinco capas con observabilidad transversal.

El núcleo lo constituyen cinco agentes basados en GPT-4.1-mini (Azure OpenAI): Synthetic Planner (traduce el contrato en un plan), Semantic Profiler (extrae expectativas semánticas), Semantic Analyzer (detecta problemas de realismo), Dataset Diagnoser (convierte hallazgos en targets) y Synthetic Enricher (genera código Python correctivo). Las salidas estructuradas mediante esquemas Pydantic eliminan por construcción los problemas de parseo de texto libre.

El flujo sigue cuatro fases —generación, análisis, diagnóstico-enriquecimiento por rondas y finalización— con bucles de retroalimentación. La política de rollback basada en fingerprints de issues garantiza que ningún cambio empeora el dataset respecto a su estado anterior.

4. Resultados

El sistema se ha evaluado sobre una batería de diez contratos ODCS de complejidad creciente, que cubren tanto contratos sintéticos diseñados para estresar características estructurales concretas (PKs y FKs compuestas, self-references, jerarquías, dominios saturados, bridges con múltiples FKs, coherencia estadística inter-tabla) como contratos realistas tomados de repositorios públicos, incluyendo el contrato AdventureWorks de Microsoft con aproximadamente cincuenta tablas y claves foráneas implícitas no declaradas formalmente. Sobre los diez contratos, el sistema ha alcanzado convergencia estructural en menos de tres rondas del bucle iterativo, sin violaciones de claves foráneas ni colisiones de claves primarias por saturación de dominio.

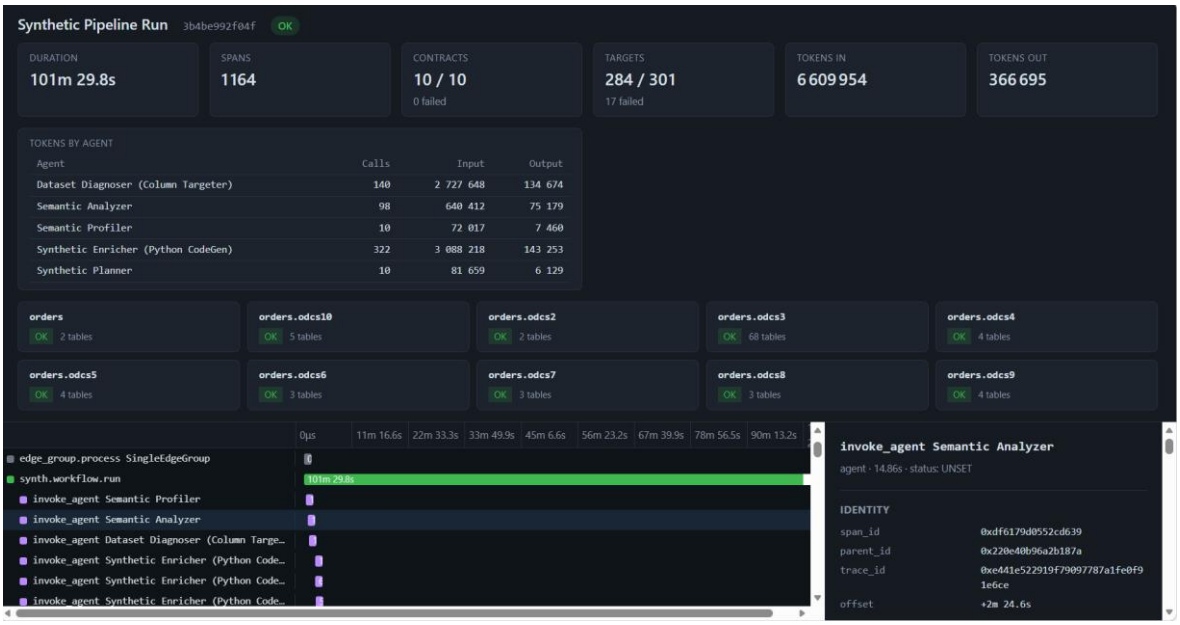

Ilustración 2 — Visor HTML del run de evaluación sobre los diez contratos.

El consumo de tokens por ejecución se sitúa típicamente en el rango de 50.000 a 200.000 tokens, lo que a precios de GPT-4.1-mini equivale a céntimos de euro por run. La aplicación práctica de las GenAI Semantic Conventions a un sistema multi-agente con bucle de retroalimentación —no documentada hasta donde alcanza la revisión bibliográfica realizada— constituye una de las aportaciones técnicas del trabajo, junto con la política de rollback por fingerprints y la propia arquitectura de cooperación entre código determinista y agentes LLM.

5. Conclusiones

El trabajo demuestra que la combinación de contratos formales como entrada generativa, código determinista para la materialización estructural y agentes LLM acotados a tareas de razonamiento semántico es una arquitectura viable, económicamente sostenible y cualitativamente superior a las alternativas existentes para la generación de datos sintéticos en entornos industriales. Las trazas estándar OpenTelemetry y el visor HTML autocontenido convierten el sistema en una herramienta auditable y reproducible.

Como trabajos futuros se identifican la extensión a otros estándares de contratos, la independencia respecto al proveedor LLM, la generación incremental y la integración con catálogos de datos.

6. Referencias

[1] Bitol Foundation / LF AI & Data. "Open Data Contract Standard (ODCS) v3.x". GitHub, 2024. https://github.com/bitol-io/open-data-contract-standard.

[9] OpenTelemetry Authors. "Semantic Conventions for Generative AI Systems". OpenTelemetry Specification, 2024. https://opentelemetry.io/docs/specs/semconv/gen-ai/.

[13] Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo, de 27 de abril de 2016 (Reglamento General de Protección de Datos). Diario Oficial de la Unión Europea, L 119, 4 de mayo de 2016. https://eur-lex.europa.eu/legal-content/ES/TXT/?uri=CELEX:32016R0679.

Índice de la memoria

Capítulo 1. Introducción 5

1.1 Motivación del proyecto 5

1.2 Descripción de las tecnologías 7

Capítulo 2. Definición del Trabajo 10

2.1 Objetivos 10

2.1.1 Objetivo general 10

2.1.2 Objetivos específicos 10

2.1.3 Criterios de éxito 12

2.2 Metodología 12

2.3 Planificación y estimación económica 14

2.3.1 Cronograma 14

2.3.2 Estimación económica 15

Capítulo 3. Estado del Arte 17

3.1 Generación de datos sintéticos 17

3.2 Data Contracts 19

3.3 Sistemas multi-agente con LLMs 20

3.4 Observabilidad de aplicaciones LLM 21

Capítulo 4. Sistema Desarrollado 23

4.1 Análisis del Sistema 23

4.1.1 Casos de uso 23

4.1.2 Requisitos funcionales 24

4.1.3 Requisitos no funcionales 25

4.1.4 Restricciones técnicas 25

4.2 Diseño 26

4.2.1 Arquitectura general 27

4.2.2 Workflow de agentes 28

4.2.3 Modelos de datos 29

4.2.4 Decisiones arquitectónicas 30

4.3 Implementación 32

4.3.1 Parser y modelo de contrato ... *32*
4.3.2 Generador determinista ... *33*
4.3.3 Agentes LLM ... *37*
4.3.4 Bucle iterativo de auto-corrección ... *43*
4.3.5 Observabilidad ... *49*

Capítulo 5. Análisis de Resultados ... ***57***
5.1 Diseño del experimento ... 57
5.1.1 Contratos de evaluación ... *57*
5.1.2 Configuración de ejecución ... *59*
5.2 Resultados estructurales ... 60
5.2.1 Convergencia y validez ... *60*
5.2.2 Comportamiento del bucle iterativo ... *60*
5.3 Resultados de calidad semántica ... 61
5.4 Coste y rendimiento ... 63
5.4.1 Consumo de tokens por agente ... *63*
5.4.2 Coste económico ... *64*
5.4.3 Rendimiento temporal ... *65*
5.5 Observabilidad en acción ... 66
5.6 Discusión ... 66

Capítulo 6. Conclusiones y Trabajos Futuros ... ***69***
6.1 Conclusiones ... 69
6.1.1 Cumplimiento de objetivos ... *69*
6.1.2 Aportaciones principales ... *70*
6.1.3 Lecciones aprendidas ... *71*
6.2 Trabajos futuros ... 72

Capítulo 7. Bibliografía ... ***74***

Índice de figuras

Ilustración 1 — Arquitectura del sistema en cinco capas con observabilidad transversal. . 10
Ilustración 2 — Visor HTML del run de evaluación sobre los diez contratos. 11
Ilustración 3 — Cronograma del proyecto con las cuatro fases de desarrollo. 15
Ilustración 4 — Diagrama UML de casos de uso del sistema. 24
Ilustración 5 — Capas funcionales, agentes y observabilidad transversal del sistema. 27
Ilustración 6 — Diagrama de secuencia entre el PlannerExecutor y el OptimizerExecutor29
Ilustración 7 — Flujo del generador determinista con sus dos ramas y la fase de materialización..... 36
Ilustración 8 — Los cinco agentes LLM con sus inputs, outputs y orden temporal. 43
Ilustración 9 — Diagrama de estados del bucle iterativo de auto-corrección. 45
Ilustración 10 — Solapamiento entre la generación de planes (prefetch) y la aplicación secuencial sobre los CSVs. 49
Ilustración 11 — Detalle de un span en el visor HTML con sus Span Events correlacionados. 53
Ilustración 12 — Visor HTML con un run completo del sistema mostrando los LLM spans. 56
Ilustración 13 — Captura del visor HTML mostrando la cabecera del run con las métricas globales y los diez contratos exitosos. 59
Ilustración 14 — Diagrama mostrando 284 targets OK frente a 17 fallidos 61
Ilustración 15 — Tokens totales consumidos por cada agente durante la ejecución de evaluación. 65

Índice de tablas

Tabla 1 — Catálogo de contratos de evaluación. 58
Tabla 2 — Consumo de tokens por agente durante el run de evaluación. 64

Capítulo 1. INTRODUCCIÓN

Este capítulo introduce el contexto del proyecto, la problemática que motiva su desarrollo y las tecnologías sobre las que se sustenta. Su objetivo es proporcionar al lector el marco conceptual mínimo para abordar los capítulos posteriores, en los que se detallan los objetivos del trabajo, el estado del arte, el sistema desarrollado y los resultados obtenidos.

1.1 MOTIVACIÓN DEL PROYECTO

La capacidad de disponer de datos representativos en entornos no productivos —desarrollo, pruebas, demostraciones internas, training de modelos de aprendizaje automático— es una necesidad transversal en cualquier organización que trabaje con datos. Sin embargo, la utilización de datos reales en estos entornos se ha vuelto cada vez más problemática durante la última década, por una combinación de factores regulatorios, técnicos y operativos que conviene desgranar.

En el plano regulatorio, el Reglamento General de Protección de Datos de la Unión Europea, vigente desde 2018 [13], impone restricciones severas al tratamiento de datos personales fuera de su finalidad declarada. Replicar una base de datos productiva sobre un entorno de desarrollo, una práctica habitual hasta hace pocos años, expone a la organización a riesgos legales relevantes: minimización inadecuada, ausencia de base legítima para el tratamiento secundario y exposición de datos a actores con privilegios de acceso desproporcionados (el equipo de desarrollo no debería ver los datos personales de un cliente real solo porque está depurando un bug). Marcos legales análogos en otras jurisdicciones —LGPD en Brasil, CCPA en California, POPIA en Sudáfrica— han generado una situación global en la que el acceso operativo a datos reales es una excepción que requiere justificación, no la norma por defecto.

En el plano técnico, los procesos de anonimización y enmascaramiento que tradicionalmente se utilizaban para mitigar este problema han demostrado ser insuficientes. Trabajos

académicos ya clásicos sobre re-identificación —desde el caso del Netflix Prize en 2008 hasta los estudios sobre datos sanitarios anonimizados— han establecido que la anonimización de un conjunto de datos suficientemente rico es prácticamente imposible si se mantiene su utilidad analítica. La consecuencia es que las técnicas de enmascaramiento, aplicadas con suficiente rigor para proporcionar garantías de privacidad reales, degradan la utilidad de los datos hasta el punto de hacerlos inservibles para los propósitos que los necesitaban.

En el plano operativo, la simple logística de obtener una copia productiva —solicitar permisos, esperar aprobaciones, configurar entornos aislados, gestionar el ciclo de vida— se ha convertido en una fricción significativa. Los equipos de ingeniería que necesitan iterar rápidamente en entornos de desarrollo no pueden permitirse esperar semanas a una copia anonimizada del entorno de producción cada vez que cambia el esquema.

La respuesta natural a esta situación es la generación de datos sintéticos: datos artificiales que comparten propiedades estadísticas y estructurales con los datos reales, pero que no derivan de ningún individuo real y por tanto no presentan riesgo de privacidad. Este enfoque ha generado un mercado creciente, con soluciones académicas, open-source y comerciales que se revisan con detalle en el Capítulo 3. Sin embargo, una observación cuidadosa de las herramientas existentes revela una brecha que el presente trabajo aborda explícitamente.

Las soluciones actuales se polarizan en dos extremos. En un extremo están las herramientas basadas en reglas explícitas —Faker, Mockaroo y similares—, que son simples, predecibles y baratas, pero ignoran completamente la estructura relacional de los datos y producen valores semánticamente irreales. En el otro extremo están los modelos generativos —SDV, CTGAN, soluciones comerciales propietarias— que producen datos estadísticamente realistas pero requieren datos reales como entrenamiento (problema de cold start y, paradójicamente, de privacidad), son opacos en sus garantías y no integran de forma nativa los contratos formales de datos que los equipos de ingeniería de datos publican junto con sus data products.

Esta brecha define la oportunidad del trabajo. Si una organización ya invierte en publicar contratos formales de sus datasets, ese contrato contiene gran parte de la información necesaria para generar datos sintéticos válidos: tipos, restricciones, relaciones, dominios, reglas de calidad. Lo que falta es la pieza de razonamiento semántico que decide, por ejemplo, si un código de producto debe parecer "P-12345" o "ABC-2024-XYZ", y esa pieza puede aportarla un agente basado en LLM. Un sistema que combine ambos elementos —rigor estructural derivado del contrato y razonamiento semántico delegado al LLM— produciría datasets más fieles, más auditables y más baratos que las alternativas existentes.

El presente Trabajo Fin de Máster materializa esta intuición en un sistema concreto, funcional y observado, cuya arquitectura, implementación y evaluación se detallan en los capítulos siguientes.

1.2 DESCRIPCIÓN DE LAS TECNOLOGÍAS

El sistema desarrollado se apoya en cinco familias tecnológicas que conviene presentar brevemente en este punto. Una revisión más detallada del estado del arte de cada una se realiza en el Capítulo 3.

La primera es el Open Data Contract Standard (ODCS) [1], un estándar abierto en formato YAML para la descripción formal de data products, mantenido por la Bitol Foundation dentro del proyecto LF AI & Data de la Linux Foundation. ODCS permite describir el esquema de un dataset (tablas, columnas, tipos lógicos y físicos), sus restricciones (claves primarias, foráneas, dominios enumerados, patrones), sus reglas de calidad (no-nulidad, recuentos esperados, relaciones de agregación) y su contexto organizativo (equipo responsable, propósito, tags semánticos). Su elección como entrada canónica del sistema responde a tres factores: es un estándar abierto y versionado, su modelo de datos cubre todo lo que el sistema necesita para generar y validar, y su adopción en la industria está creciendo significativamente desde 2023.

La segunda es el Microsoft Agent Framework (MAF) [7], un framework de orquestación de agentes basados en LLM publicado por Microsoft en 2024 como evolución directa de

Semantic Kernel y AutoGen. MAF proporciona abstracciones de alto nivel —ChatAgent, Workflow, Executor, WorkflowContext— que permiten construir sistemas multi-agente con paso de mensajes tipados y estados compartidos, a la vez que se mantiene el control fino sobre la concurrencia y la observabilidad. Una característica decisiva de MAF para este trabajo es su emisión nativa de trazas conformes a las GenAI Semantic Conventions de OpenTelemetry [9], lo que evita tener que instrumentar manualmente cada invocación al modelo.

La tercera es la familia de modelos GPT-4.1-mini servidos a través de Azure OpenAI [8]. La elección de este modelo, frente a alternativas como GPT-4o, Claude Sonnet o Llama 3 alojado localmente, responde a un compromiso entre capacidad y coste operativo. GPT-4.1-mini ofrece structured outputs nativos basados en JSON Schema, una ventana de contexto de 128.000 tokens y un coste aproximadamente diez veces inferior a los modelos de gama alta. Para un sistema que invoca al modelo entre 30 y 50 veces por run, esta diferencia de coste convierte un proyecto inviable en un proyecto sostenible.

La cuarta es OpenTelemetry, el estándar abierto de la Cloud Native Computing Foundation (CNCF) para la instrumentación, generación y exportación de telemetría —trazas, métricas y logs— en sistemas distribuidos. Su elección frente a alternativas comerciales especializadas en LLMOps (Langfuse, Phoenix, LangSmith) responde al principio de no acoplar el código del sistema a un backend concreto: una traza emitida en formato OpenTelemetry puede consumirse desde Jaeger, Tempo, Honeycomb, otel-desktop-viewer o un visor ad-hoc sin modificar el código instrumentado. Las GenAI Semantic Conventions, publicadas por la OpenTelemetry Specification en 2024, definen las convenciones de nombrado para atributos específicos del dominio LLM (gen_ai.usage.input_tokens, gen_ai.request.model, etc.) que el sistema utiliza tanto para registrar su propio comportamiento como para interpretar las trazas que MAF emite por su cuenta.

La quinta es Python 3.10+ como lenguaje de implementación, junto con un conjunto reducido pero significativo de bibliotecas: agent-framework como infraestructura base del sistema multi-agente, Pydantic para los esquemas de salida estructurada de los agentes y

para los modelos de datos internos, PyYAML para el análisis sintáctico de los contratos ODCS, OR-Tools de Google para el solucionador CP-SAT empleado en la generación de tablas con dependencias cíclicas, pandas para la manipulación de los CSV en las fases de generación y enriquecimiento, Faker para la generación de valores semánticamente realistas (nombres, direcciones, etc.) durante el enriquecimiento, rstr para la generación de cadenas conformes a expresiones regulares, y OpenTelemetry para la trazabilidad y observabilidad del pipeline. La justificación de cada uno de estos componentes y su rol concreto en el sistema se desarrolla en el Capítulo 4.

Capítulo 2. DEFINICIÓN DEL TRABAJO

Este capítulo formaliza los objetivos del Trabajo Fin de Máster, describe la metodología seguida para alcanzarlos y presenta la planificación temporal y la estimación económica del desarrollo. Mientras que el Capítulo 1 ha establecido el por qué del trabajo y el Capítulo 4 detallará el cómo, el presente capítulo responde al qué: qué se ha pretendido lograr, bajo qué criterios se ha considerado logrado y con qué recursos se ha llevado a cabo.

2.1 OBJETIVOS

2.1.1 OBJETIVO GENERAL

El objetivo general del trabajo es diseñar e implementar un sistema basado en agentes LLM capaz de generar datasets sintéticos a partir de un Open Data Contract, garantizando la validez estructural respecto al contrato y la plausibilidad semántica de los valores generados, con observabilidad estándar OpenTelemetry y un nivel de coste operativo compatible con su uso recurrente.

Este objetivo es deliberadamente compuesto: incorpora cuatro dimensiones que el sistema debe satisfacer simultáneamente, no de forma alternativa. Un sistema que produjera datos válidos pero no observables sería inadecuado para entornos de producción; uno observable pero costoso sería inadecuado para uso recurrente; uno barato pero estructuralmente inválido sería inútil para cualquier propósito serio.

2.1.2 OBJETIVOS ESPECÍFICOS

El objetivo general se descompone en seis objetivos específicos, cada uno de los cuales se aborda en una sección concreta del Capítulo 4 y se evalúa en el Capítulo 5.

El objetivo O1 consiste en desarrollar un parser robusto del Open Data Contract Standard capaz de procesar contratos en formato YAML conformes a la especificación oficial y,

además, tolerante a las variaciones menores que se observan habitualmente en contratos de producción: formatos alternativos para declarar relaciones (cadena, lista, diccionario), secciones no canónicas, custom properties con estructura inesperada. El parser debe traducir el contrato a una representación interna inmutable consumible por el resto del sistema, preservando las secciones que no mapea explícitamente.

El objetivo O2 consiste en construir un generador determinista que materialice un contrato en filas concretas respetando todas las restricciones estructurales declaradas: claves primarias simples y compuestas, claves foráneas simples, compuestas y con auto-referencia (self-FK), tipos lógicos del estándar (UUID, integer con rangos, fecha, timestamp, cadena con patrón regex y restricciones de longitud), dominios enumerados y reglas de calidad. El generador debe resolver correctamente los casos topológicamente complejos —grupos cíclicos de tablas, dominios saturables— mediante mecanismos especializados (algoritmo de Tarjan para componentes fuertemente conexos, solucionador CP-SAT para asignaciones bajo restricciones).

El objetivo O3 consiste en implementar una capa de análisis multi-dimensional combinando cuatro perspectivas complementarias: extracción semántica de expectativas a partir del contrato, análisis estadístico determinista de cardinalidades y distribuciones, análisis de coherencia inter-columna e inter-tabla basado en reglas declarativas, y análisis semántico interpretativo basado en LLM para detectar problemas de realismo que las heurísticas previas no pueden capturar.

El objetivo O4 consiste en desarrollar un bucle iterativo de auto-corrección que convierta los hallazgos del análisis en targets accionables y los procese mediante un agente LLM especializado en la generación de código correctivo, con una política de rollback basada en fingerprints de issues que garantice que ningún cambio del sistema empeora el dataset respecto a su estado anterior.

El objetivo O5 consiste en integrar observabilidad estándar OpenTelemetry sobre el pipeline, implementando los tres pilares de la especificación —trazas, eventos correlacionados y

métricas— de forma que la telemetría emitida sea consumible por cualquier backend OTel sin modificación del código y siga las GenAI Semantic Conventions publicadas en 2024.

El objetivo O6 consiste en desarrollar un visor HTML autocontenido que permita el análisis de las trazas generadas por una ejecución, sin requerir servidor, instalación de dependencias ni acceso a backends externos, y que sea reutilizable de forma independiente sobre cualquier directorio de run histórico.

2.1.3 CRITERIOS DE ÉXITO

Cada objetivo específico tiene asociados criterios de éxito concretos. Para O1, el parser debe procesar correctamente una batería de diez contratos de prueba que cubre un espectro amplio de casos: contratos sintéticos diseñados para estresar características concretas del sistema (PKs compuestas, FKs compuestas, self-references, dominios saturables, jerarquías, bridges con múltiples FKs) y contratos realistas tomados de repositorios públicos, incluyendo el contrato AdventureWorks de Microsoft con ~50 tablas y FKs implícitas no declaradas formalmente. Para O2, el generador debe producir datasets de hasta un millón de filas sin violaciones de FK ni colisiones de PK por saturación, sobre los contratos de evaluación. Para O3, la capa de análisis debe detectar al menos los problemas de realismo introducidos deliberadamente en los datasets de prueba (cadenas sin sentido, valores fuera de dominio, distribuciones planas en columnas con sesgo esperado). Para O4, el bucle debe converger en un máximo de tres rondas y no debe introducir regresiones tras un rollback. Para O5, las trazas generadas deben ser interpretables por al menos dos backends OTel distintos sin modificación. Para O6, el visor debe abrirse a doble click sin servidor y mostrar correctamente la jerarquía de spans de un run completo. La validación cuantitativa de cada criterio se realiza en el Capítulo 5.

2.2 METODOLOGÍA

El trabajo se ha desarrollado siguiendo una metodología iterativa-incremental, organizada en ciclos cortos de implementación, validación y refactorización. Cada ciclo se inició con un caso de prueba concreto —típicamente un nuevo contrato o un escenario estructural

particular— y avanzó hasta que ese caso quedó cubierto por el sistema, momento en el que se incorporaba un nuevo caso y se reiniciaba el ciclo.

Esta forma de trabajo, frente a una alternativa de tipo waterfall en la que se diseña todo el sistema antes de implementar nada, se justifica por la naturaleza del problema: gran parte de las decisiones técnicas no eran predecibles desde el papel. Decisiones como la división entre código determinista y agentes LLM, la política de rollback basada en fingerprints o la sustitución del fichero de métricas por una derivación a posteriori desde las trazas son consecuencia directa de problemas concretos encontrados durante la implementación, no de un diseño previo. Una metodología iterativa permite que estas decisiones emerjan de forma orgánica.

La filosofía de diseño que ha guiado todas las decisiones se puede resumir en tres principios complementarios. El primero es "el LLM solo decide lo que no puede decidirse determinísticamente": cualquier tarea que pueda implementarse con código Python convencional se implementa así, y solo se delega al modelo lo que requiere razonamiento semántico. Este principio reduce coste, aumenta reproducibilidad y elimina por construcción una clase entera de errores. El segundo es "todo cambio sobre el dataset debe ser reversible": ninguna operación del sistema sobre los CSV es destructiva sin un snapshot previo, lo que permite el rollback automático y, secundariamente, facilita la depuración. El tercero es "la observabilidad es parte del sistema, no un añadido": las trazas se diseñan junto con la lógica que las emite, no se instrumentan a posteriori. Este tercer principio, en particular, ha demostrado ser el más rentable durante el desarrollo: la mayor parte de los problemas no triviales que surgieron se diagnosticaron leyendo trazas en el visor.

La estrategia de testing ha sido pragmática. No se han desarrollado tests unitarios automatizados al estilo xUnit, ya que gran parte del comportamiento del sistema depende de invocaciones reales al LLM cuya naturaleza estocástica las hace inadecuadas para aserciones binarias. En su lugar, se ha empleado una batería de contratos de prueba que actúan como test fixtures a nivel de sistema completo: cada contrato ejercita un conjunto específico de capacidades (composite FKs, ciclos, self-references, dominios saturables, reglas de calidad

complejas) y se utiliza como caso de regresión cuando se modifica alguna parte del sistema. La validación se realiza comparando los CSV generados con el contrato declarado mediante el validador interno del propio sistema.

2.3 PLANIFICACIÓN Y ESTIMACIÓN ECONÓMICA

2.3.1 CRONOGRAMA

El trabajo se ha desarrollado durante un período aproximado de tres meses, organizados en cuatro fases principales que se solaparon parcialmente.

La fase de fundamentos (mes 1) cubrió la revisión bibliográfica, el estudio de los frameworks candidatos (LangGraph, AutoGen, MAF) y la elección final del stack técnico. El resultado de esta fase fue el documento de propuesta de TFM y un prototipo desechable de un solo agente, sin generador determinista, que se utilizó únicamente para validar las capacidades del MAF.

La fase de núcleo determinista (meses 1-2) cubrió la implementación del parser ODCS y del generador determinista. Esta fase fue la más densa en líneas de código y la más compleja desde el punto de vista algorítmico, ya que en ella se resolvieron los problemas de detección de SCCs, integración con el solver CP-SAT y los tres patrones especiales de FK descritos en el Capítulo 4. Al final de esta fase el sistema era capaz de generar datasets estructuralmente válidos pero sin ninguna forma de análisis ni corrección posterior.

La fase de agentes y bucle iterativo (meses 2-3) cubrió la implementación de los cinco agentes LLM, la capa de análisis, el Diagnoser, el Enricher y el bucle de auto-corrección con su política de rollback. Esta fue la fase más rica en decisiones de diseño y en aprendizajes; gran parte de los principios de la metodología descrita en la sección anterior se consolidaron durante este período.

La fase de observabilidad y consolidación (mes 3) cubrió la integración de OpenTelemetry, la coexistencia con el TracerProvider de MAF, los Span Events para la correlación de logs

y el desarrollo del visor HTML. Esta fase incluyó también la batería de pruebas finales, la redacción de la memoria y la preparación de los artefactos de evaluación.

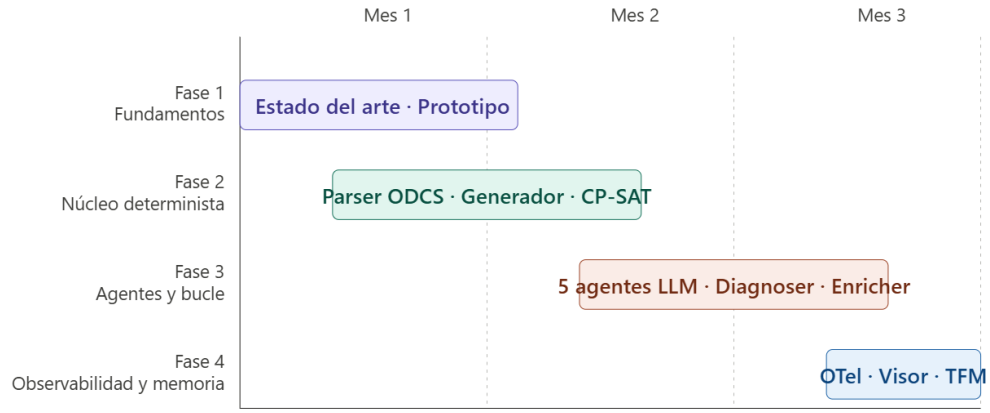

Ilustración 3 — Cronograma del proyecto con las cuatro fases de desarrollo.

2.3.2 ESTIMACIÓN ECONÓMICA

El coste económico del proyecto se descompone en dos partidas: el coste de desarrollo —fundamentalmente horas de trabajo del autor— y el coste operativo —fundamentalmente consumo de la API de Azure OpenAI durante el desarrollo y la evaluación.

En cuanto al coste de desarrollo, considerando una dedicación media de 25 horas semanales durante 12 semanas (tres meses), el esfuerzo total asciende a aproximadamente 300 horas. Aplicando una tarifa de mercado para un perfil de ingeniería de datos junior con conocimientos de IA de 17 €/hora, el coste de desarrollo equivalente sería de 5.100 €. Esta cifra debe interpretarse como coste de oportunidad, no como coste real, dado que se trata de un trabajo académico.

En cuanto al coste operativo, el consumo de la API de Azure OpenAI durante toda la fase de desarrollo y evaluación se estima en torno a 15-20 €. Esta cifra, deliberadamente baja, es uno de los resultados clave del trabajo: la arquitectura de cooperación entre código determinista y agentes LLM, descrita en el Capítulo 4, mantiene el consumo de tokens dentro de un orden de magnitud que permite ejecutar el sistema cientos de veces durante el

desarrollo sin un coste prohibitivo. Una ejecución completa sobre un contrato no trivial consume típicamente entre 50.000 y 200.000 tokens, lo que a precios de GPT-4.1-mini en abril de 2026 equivale a céntimos de euro por run.

A las dos partidas anteriores se suma un coste menor de infraestructura local (estación de trabajo, almacenamiento) y de licencias de software (todo el stack utilizado es open-source salvo Azure OpenAI, que es de pago por uso), que no se cuantifica explícitamente al ser parte del entorno general del autor.

El coste total estimado del proyecto, sumando coste de oportunidad de desarrollo y coste operativo real, se sitúa en torno a los 5.100 -5.200 €. En un escenario de despliegue del sistema como servicio recurrente, el coste marginal por dataset generado sería el coste operativo de tokens más el coste de cómputo local (despreciable a la escala del run), lo que sitúa el modelo en un rango competitivo frente a las alternativas comerciales revisadas en el Capítulo 3.

Capítulo 3. ESTADO DEL ARTE

Este capítulo revisa el contexto técnico y académico en el que se inscribe el trabajo realizado. La revisión se estructura en cuatro áreas que confluyen en el sistema desarrollado: las técnicas existentes para la generación de datos sintéticos, el movimiento de los data contracts como estándar emergente de gobierno de datos, los frameworks contemporáneos para la construcción de sistemas multi-agente basados en Large Language Models, y la observabilidad aplicada específicamente a aplicaciones LLM. Cada una de estas áreas se cubre con la profundidad necesaria para situar las decisiones técnicas tomadas a lo largo del Capítulo 4 y para señalar las limitaciones de las aproximaciones existentes que motivan el presente trabajo.

3.1 GENERACIÓN DE DATOS SINTÉTICOS

La generación sintética de datos es una disciplina con varias décadas de recorrido, pero su relevancia industrial ha crecido significativamente en los últimos diez años por dos vectores convergentes: las regulaciones de privacidad —GDPR en la Unión Europea, CCPA en California, LGPD en Brasil— que restringen el uso de datos reales en entornos de desarrollo y testing, y el auge del aprendizaje automático, que demanda volúmenes de datos cada vez mayores para entrenamiento y evaluación. Las aproximaciones existentes pueden agruparse en tres familias técnicamente distintas.

La primera familia, históricamente la más antigua, es la generación basada en reglas. Bibliotecas como Faker, Mockaroo o Mimesis proporcionan catálogos de generadores predefinidos —nombres, direcciones, números de teléfono, códigos postales— que el desarrollador combina manualmente para componer filas. Su ventaja es la simplicidad y la previsibilidad: el comportamiento del generador es completamente transparente y reproducible. Sus limitaciones, sin embargo, son severas para escenarios reales. Estas herramientas no entienden la estructura relacional de los datos: generar una clave foránea

coherente entre dos tablas requiere código ad-hoc escrito por el usuario. Tampoco entienden la semántica del dominio: un código de producto generado por Faker es indistinguible de un identificador genérico, y replicar las distribuciones reales de un dominio (la curva pareto típica de transacciones, la estacionalidad de las ventas) requiere implementación manual.

La segunda familia es la generación basada en modelos estadísticos. El proyecto académico de referencia en este espacio es el Synthetic Data Vault (SDV) [12], que aplica modelos probabilísticos —cópulas gaussianas, modelos jerárquicos bayesianos, redes generativas adversarias tabulares como CTGAN y TVAE [16]— para aprender la distribución conjunta de un dataset real y muestrear datos sintéticos que la replican. Estas aproximaciones resuelven el problema de la fidelidad estadística: las distribuciones marginales y las correlaciones entre columnas se preservan razonablemente bien. Pero introducen tres problemas nuevos. Primero, requieren datos reales como entrada de entrenamiento, lo que en muchos contextos industriales no es posible (precisamente por las regulaciones que motivan el uso de datos sintéticos en primer lugar). Segundo, son opacos: el comportamiento del modelo no se puede auditar fácilmente, y las garantías sobre las propiedades del dataset generado son probabilísticas, no determinísticas. Tercero, no respetan contratos formales: un CTGAN entrenado sobre una tabla puede perfectamente generar valores que violan restricciones declarativas (rangos, patrones, claves foráneas) si esas restricciones no estaban presentes de forma consistente en los datos de entrenamiento.

La tercera familia, la más reciente, es la generación basada en LLMs. Trabajos académicos recientes como GReaT [2] (Generation of Realistic Tabular data, Borisov et al. 2023) o TabuLa (Zhao et al. 2023) demuestran que un LLM ajustado adecuadamente puede generar filas tabulares de calidad competitiva con CTGAN. Las herramientas comerciales del espacio —Mostly AI, Tonic.ai, Gretel.ai— combinan típicamente modelos generativos con post-procesado y validación, ofreciendo soluciones cerradas y de alta calidad. Sin embargo, estas aproximaciones siguen presentando dos limitaciones que el sistema desarrollado en este trabajo aborda explícitamente. Primera: el coste por fila es proporcional al tamaño del dataset, lo que las hace económicamente inviables para volúmenes de millones de filas con presupuestos de proyecto razonables. Segunda: la integración con contratos formales es ad-

hoc cuando existe; ninguna de las soluciones revisadas adopta ODCS o un estándar similar como entrada canónica.

El presente trabajo se distingue de las tres familias anteriores en que combina lo mejor de cada una: el rigor estructural de las reglas (mediante el generador determinista que respeta el contrato por construcción), la flexibilidad semántica de los LLMs (limitada a las decisiones que el código no puede tomar) y la auditoría completa que ninguna de las tres ofrece nativamente (mediante OpenTelemetry).

3.2 DATA CONTRACTS

El término data contract describe un acuerdo formal y versionado entre el productor y los consumidores de un data product, en el que se especifican el esquema de los datos, sus restricciones de calidad, sus garantías de servicio (frecuencia de actualización, latencia) y las políticas de cambio. La idea, aunque tiene precedentes en el mundo de las APIs (OpenAPI Specification, gRPC) y de los esquemas de mensajes (Avro, Protobuf), se popularizó en el espacio de datos a partir de 2022 mediante artículos influyentes publicados por equipos de PayPal, GoCardless y Convoy [11], en los que se describía el uso de contratos como mecanismo para reducir la fragilidad de los pipelines de datos en arquitecturas de tipo data mesh.

A nivel de estándares, dos especificaciones compiten actualmente por la adopción industrial. La primera es el Open Data Contract Standard (ODCS) [1], un estándar abierto mantenido por la Bitol Foundation (proyecto de la Linux Foundation AI & Data) y actualmente en su versión 3.x. ODCS adopta un formato YAML con secciones declarativas para esquema, calidad, equipo responsable, propósito y custom properties, y se distingue por su orientación explícita a data products en arquitecturas distribuidas. La segunda es la Data Contract Specification (datacontract.com), de origen más comercial, con una sintaxis similar pero con un enfoque más orientado a APIs de datos.

Ambos estándares comparten una limitación de cara al presente trabajo: están diseñados como contratos runtime (validar que los datos producidos cumplen el contrato), no como

contratos generativos (producir datos que satisfagan el contrato). El sistema desarrollado adopta ODCS como entrada canónica precisamente porque interpreta el contrato no como una restricción a verificar a posteriori, sino como una especificación generativa de la que derivar los datos sintéticos. Esta inversión de la perspectiva no está, hasta donde alcanza la revisión bibliográfica realizada, documentada en otras herramientas del ecosistema.

3.3 SISTEMAS MULTI-AGENTE CON LLMS

La construcción de sistemas en los que varios agentes basados en LLMs cooperan para resolver tareas complejas es un área de investigación y desarrollo extremadamente activa. Tres frameworks concentran la mayor parte de la adopción industrial en 2025-2026:

- LangGraph (LangChain), basado en el modelo de grafos de estados sobre el que se ejecutan los agentes; ofrece gran flexibilidad de orquestación y un ecosistema de herramientas maduro.
- AutoGen (Microsoft Research), centrado en patrones de conversación entre agentes, con énfasis en escenarios de tipo peer-to-peer en los que múltiples agentes debaten o se complementan.
- CrewAI, orientado a una abstracción más alta de "rol" y "tarea", con un público objetivo menos técnico.

A estos se suma el más reciente Microsoft Agent Framework (MAF) [7], lanzado en 2024 como evolución directa del Semantic Kernel y de AutoGen. MAF se distingue por dos características relevantes para este trabajo: ofrece una abstracción de workflow basada en executors y mensajes tipados que es notablemente más estructurada que la de sus competidores, y emite trazas conformes a las GenAI Semantic Conventions de OpenTelemetry de forma nativa, sin instrumentación adicional. Esta segunda característica fue decisiva en la elección del framework, ya que evitó tener que implementar manualmente la instrumentación de cada agente.

Más allá de los frameworks, la literatura académica ha consolidado varios patrones de diseño para sistemas multi-agente. El patrón ReAct [17] (Yao et al. 2022) intercala razonamiento y

acción dentro del mismo agente. El patrón Reflexión [14] (Shinn et al. 2023) introduce una etapa de auto-crítica del agente sobre su propia salida. Los patrones orquestador-trabajadores y debate describen formas de coordinar varios agentes con responsabilidades diferenciadas. El sistema desarrollado en este trabajo se inscribe claramente en el patrón orquestador-trabajadores, con una variante distintiva: el orquestador (OptimizerExecutor) no es un agente LLM, sino código Python determinista. Esta decisión, que va en contra de la tendencia general en la literatura, se justifica por las garantías de reproducibilidad y por el control fino sobre la concurrencia que un orquestador determinista ofrece.

En el ámbito específico de la ingeniería de datos, las aplicaciones documentadas de sistemas multi-agente son aún incipientes. Trabajos como Data Interpreter (MetaGPT, 2024) o DataChat exploran el uso de agentes para análisis exploratorio y generación de informes, pero no se conocen, hasta donde alcanza la revisión, sistemas multi-agente publicados que aborden específicamente la generación de datos sintéticos a partir de contratos formales.

3.4 OBSERVABILIDAD DE APLICACIONES LLM

La observabilidad aplicada a sistemas que invocan LLMs —denominada habitualmente LLMOps u observabilidad GenAI— se ha convertido en un subdominio diferenciado en los últimos dos años. La preocupación central no es solo la latencia o la disponibilidad (los problemas clásicos del APM tradicional), sino dimensiones específicas del dominio: el coste por token, la calidad de las respuestas, la deriva del comportamiento del modelo entre versiones y la trazabilidad de las decisiones tomadas por agentes.

Las herramientas comerciales del espacio se dividen en dos grupos. El primero son las plataformas verticales especializadas: Langfuse, Phoenix (Arize), LangSmith (LangChain) y Helicone ofrecen experiencias de usuario muy pulidas, dashboards orientados a agentes, anotación de calidad y A/B testing de prompts. Su valor de uso es elevado, pero presentan un acoplamiento significativo a su backend específico: cambiar de Langfuse a LangSmith implica reinstrumentar la aplicación. El segundo grupo son los backends OpenTelemetry generalistas —Jaeger, Tempo (Grafana), Honeycomb, Datadog APM— que consumen

trazas estándar OTel sin asumir particularidades del dominio LLM, perdiendo algo de especialización a cambio de portabilidad.

La pieza que ha permitido cerrar este gap entre especialización y portabilidad son las GenAI Semantic Conventions de OpenTelemetry [9], publicadas por la OpenTelemetry Specification en 2024 y actualmente en estado experimental. Estas convenciones definen una nomenclatura común para los atributos relevantes en aplicaciones LLM: gen_ai.system, gen_ai.request.model, gen_ai.usage.input_tokens, gen_ai.usage.output_tokens, gen_ai.operation.name, gen_ai.agent.name, entre otras. Una traza emitida conforme a estas convenciones es interpretable de forma idéntica por cualquier backend OTel, lo que devuelve al usuario la libertad de elegir su herramienta de visualización sin acoplarse al instrumentador.

El sistema desarrollado en este trabajo emite trazas conformes a estas convenciones —parte de ellas heredadas del MAF, parte añadidas por el código propio— y se ha verificado que son consumibles tanto por el visor HTML ad-hoc implementado en este trabajo como por otel-desktop-viewer y por scripts de análisis basados en pandas. Esta portabilidad se considera, junto con la cooperación determinista-LLM descrita en el Capítulo 4, una de las decisiones técnicas más importantes y mejor sostenidas del sistema.

Capítulo 4. SISTEMA DESARROLLADO

Este capítulo describe el sistema diseñado e implementado en el marco de este Trabajo Fin de Máster. Se presenta primero el análisis de requisitos y casos de uso, después el diseño arquitectónico con sus decisiones técnicas más relevantes, y finalmente la implementación detallada de cada componente, con especial atención a los aspectos no triviales del sistema: la cooperación entre código determinista y agentes basados en LLM, el bucle iterativo de auto-corrección, y la integración de observabilidad estándar OpenTelemetry.

4.1 ANÁLISIS DEL SISTEMA

4.1.1 CASOS DE USO

El sistema se concibe como una herramienta de línea de comandos y servicio interno orientado a equipos de ingeniería de datos. Los casos de uso principales identificados son:

- CU1 — Generar dataset de desarrollo a partir de un contrato existente. Un desarrollador necesita un volumen de datos de prueba que respete un contrato ODCS ya publicado por su equipo. El sistema toma el contrato y produce los CSV correspondientes a cada tabla declarada, validados estructuralmente.
- CU2 — Auditar trazas y costes de una ejecución. Un MLOps engineer necesita entender el comportamiento del sistema en una ejecución concreta: qué agentes se invocaron, cuántos tokens se consumieron, qué decisiones tomó cada agente. El sistema persiste trazas OpenTelemetry estándar y un visor HTML autocontenido las hace navegables.

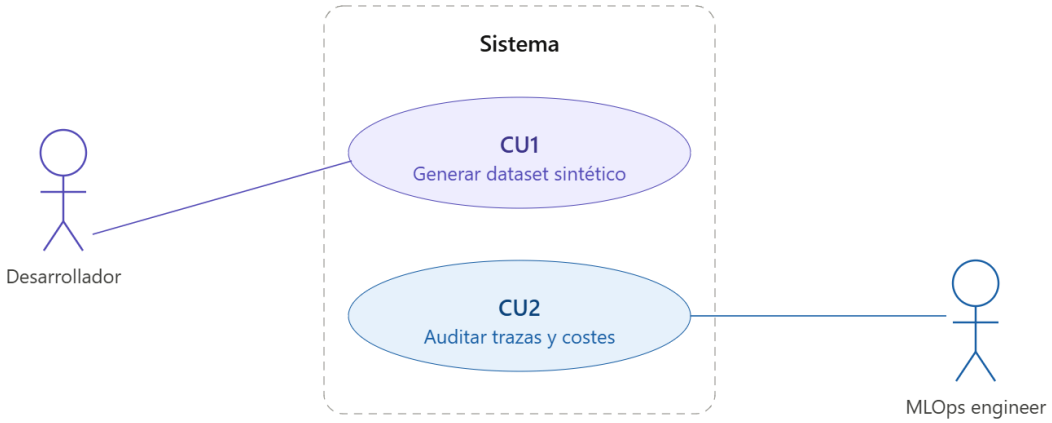

Ilustración 4 — Diagrama UML de casos de uso del sistema.

4.1.2 REQUISITOS FUNCIONALES

Del análisis de los casos de uso se derivan los siguientes requisitos funcionales:

- RF1. El sistema debe ingerir contratos en formato Open Data Contract Standard (ODCS) versión 3.x, tolerando variaciones menores en el formato.
- RF2. El sistema debe generar datos sintéticos respetando todas las restricciones estructurales declaradas: claves primarias (simples y compuestas), claves foráneas (incluidas compuestas a nivel de tabla), tipos lógicos, dominios enumerados, restricciones de longitud y patrones regex.
- RF3. El sistema debe ejecutar un análisis multi-dimensional sobre el dataset generado, que combine la extracción semántica de expectativas a partir del contrato con análisis estadístico, de coherencia entre columnas e interpretativo basado en LLM.
- RF4. El sistema debe corregir iterativamente los problemas de calidad detectados, aplicando los cambios sobre los CSV generados y revalidando tras cada modificación.

- RF5. El sistema debe persistir un artefacto autocontenido por ejecución que incluya los datos generados, los planes intermedios, las trazas, las métricas y un manifiesto JSON resumen.

4.1.3 REQUISITOS NO FUNCIONALES

- RNF1 — Reproducibilidad. Para una misma entrada (contrato + semilla), el sistema debe producir resultados deterministas en su parte estructural. La parte semántica delegada al LLM no es estrictamente reproducible, pero el sistema debe registrar las trazas suficientes para entender qué decisiones se tomaron y por qué.
- RNF2 — Observabilidad. El sistema debe emitir trazas, eventos correlacionados y métricas siguiendo el estándar OpenTelemetry, con el objetivo de integrarse con cualquier backend compatible (Jaeger, Tempo, Honeycomb) sin modificaciones de código.
- RNF3 — Tolerancia a fallos del proveedor LLM. Las llamadas a la API de Azure OpenAI son no-deterministas en latencia y pueden fallar (timeouts, rate limits, respuestas malformadas). El sistema debe degradar graciosamente: un fallo en un target concreto no debe abortar la ejecución completa.
- RNF4 — Extensibilidad. Añadir un nuevo agente, un nuevo analizador o un nuevo tipo de target debe ser una operación local que no requiera modificar el resto del pipeline.
- RNF5 — Escalabilidad horizontal del LLM. El cuello de botella real del sistema no es el cómputo local, sino el contexto del LLM. El sistema debe procesar datasets grandes (millones de filas) sin que el contexto del modelo crezca proporcionalmente.

4.1.4 RESTRICCIONES TÉCNICAS

El diseño del sistema está condicionado por tres restricciones técnicas de naturaleza distinta:

- La primera es la ventana de contexto del LLM. GPT-4.1-mini ofrece 128K tokens de contexto, pero los prompts útiles raramente superan los 30-40K antes de que la calidad de la respuesta se degrade por el conocido fenómeno "lost in the middle".

Esto fuerza a procesar las tablas en batches y a no enviar nunca el dataset completo al modelo, sino muestras representativas.

- La segunda es el coste por token. A precios de mercado en abril de 2026 (≈0,40 €/M tokens de entrada y 1,60 €/M tokens de salida en GPT-4.1-mini), una ejecución sobre un contrato no trivial puede consumir entre 50K y 200K tokens. La eficiencia del prompting impacta directamente en el TCO operativo del sistema.
- La tercera es el no-determinismo del LLM. La misma entrada puede producir respuestas distintas, malformadas o no parseables. El sistema absorbe esta característica mediante structured outputs basados en Pydantic, retries con back-off, y un mecanismo de rollback que revierte cambios cuando introducen regresiones.

4.2 DISEÑO

El sistema sigue una arquitectura de canalización (pipeline) en la que un contrato ODCS atraviesa varias etapas hasta materializarse en un dataset sintético validado. La canalización está orquestada por el Microsoft Agent Framework (MAF), que coordina la ejecución de los agentes y gestiona el paso de mensajes entre ellos. Cinco agentes basados en Large Language Models cooperan con un núcleo de código determinista que realiza las operaciones que no requieren razonamiento: el parseo del contrato, la materialización de filas conforme a las restricciones estructurales, la validación cruzada con el contrato y la aplicación física de los cambios sobre los CSV. Esta separación de responsabilidades es deliberada y se justifica más adelante.

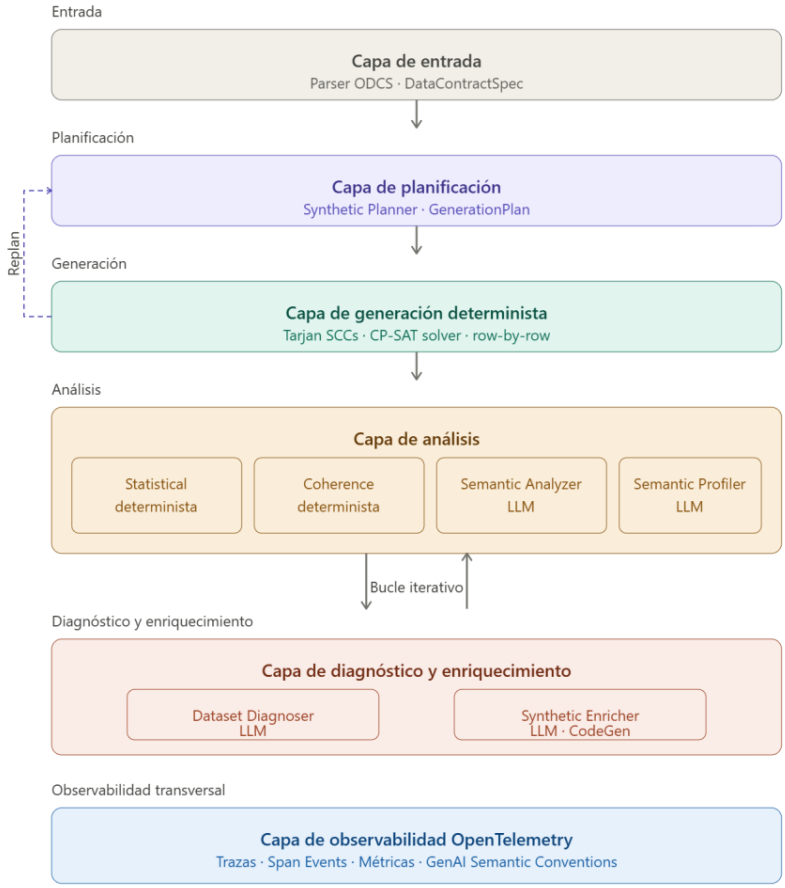

Ilustración 5 — Capas funcionales, agentes y observabilidad transversal del sistema.

4.2.1 ARQUITECTURA GENERAL

A alto nivel, el sistema se organiza en cinco capas funcionales que se comunican secuencialmente, con un bucle de retroalimentación entre la capa de validación y las capas de planificación y enriquecimiento. La capa de entrada carga el contrato ODCS desde disco, lo parsea a una representación interna basada en dataclasses de Python y lo expone al resto del sistema mediante la clase DataContractSpec. La capa de planificación, gobernada por el agente Synthetic Planner, traduce el contrato a un GenerationPlan que especifica el número de filas por tabla, los perfiles estructurales aplicables y la semilla aleatoria. La capa de generación materializa el plan en datos físicos (CSV) mediante un generador determinista que combina dos estrategias: una iteración fila a fila para tablas acíclicas y una resolución vía CP-SAT para grupos cíclicos de tablas (strongly connected components). La capa de

análisis examina el dataset producido bajo cuatro perspectivas complementarias —de perfil semántico vía LLM (que extrae las expectativas declaradas en el contrato), estadística, de coherencia inter-columna e inter-tabla, y semántica interpretativa vía LLM— y produce un AnalysisReport con los hallazgos detectados. La capa de diagnóstico y enriquecimiento convierte esos hallazgos en targets accionables, genera código Python correctivo, lo aplica sobre los CSV, y revalida el resultado. Finalmente, una capa transversal de observabilidad registra trazas, eventos y métricas siguiendo el estándar OpenTelemetry, atravesando todas las capas anteriores sin acoplarse a ninguna.

El bucle de retroalimentación es central al diseño. Cuando el generador encuentra una infeasibilidad estructural recuperable —por ejemplo, una tabla hija con más filas de las que el dominio de su clave foránea permite—, el control vuelve al Planner con el error como contexto, y se replanifica. De forma análoga, si el análisis detecta problemas de calidad en el dataset generado, el Diagnoser y el Enricher entran en juego en rondas sucesivas hasta que no quedan correcciones aplicables o se agota el presupuesto de rondas configurado.

4.2.2 WORKFLOW DE AGENTES

Internamente, el sistema utiliza dos executors MAF como nodos del grafo de ejecución: PlannerExecutor y OptimizerExecutor. El primero actúa como punto de entrada del workflow: resuelve los contratos a procesar a partir de la variable de entorno SYNTH_CONTRACT_PATH, genera un run_id único, inicializa la observabilidad para esa ejecución concreta, e invoca al agente Synthetic Planner una vez por contrato para producir un plan de generación. El OptimizerExecutor recibe el plan y ejecuta el bucle principal: generar, analizar, diagnosticar, enriquecer, validar. Esta segmentación responde a una decisión de diseño deliberada: el Planner no necesita conocer nada del bucle de corrección, y el Optimizer se beneficia de recibir un plan ya validado, lo que simplifica la lógica de retry.

Aunque conceptualmente hay cinco agentes, MAF solo los expone como recursos invocables: _get_planner_agent(), _get_diagnoser_agent(), _get_enricher_agent(), _get_profiler_agent() y _get_semantic_agent(). Cada agente está configurado mediante un fichero YAML que define su system prompt, su esquema de salida estructurada (Pydantic)

y los parámetros del modelo. Esta separación permite ajustar el comportamiento de un agente —por ejemplo, hacer al Diagnoser más conservador— sin tocar código, modificando únicamente su YAML.

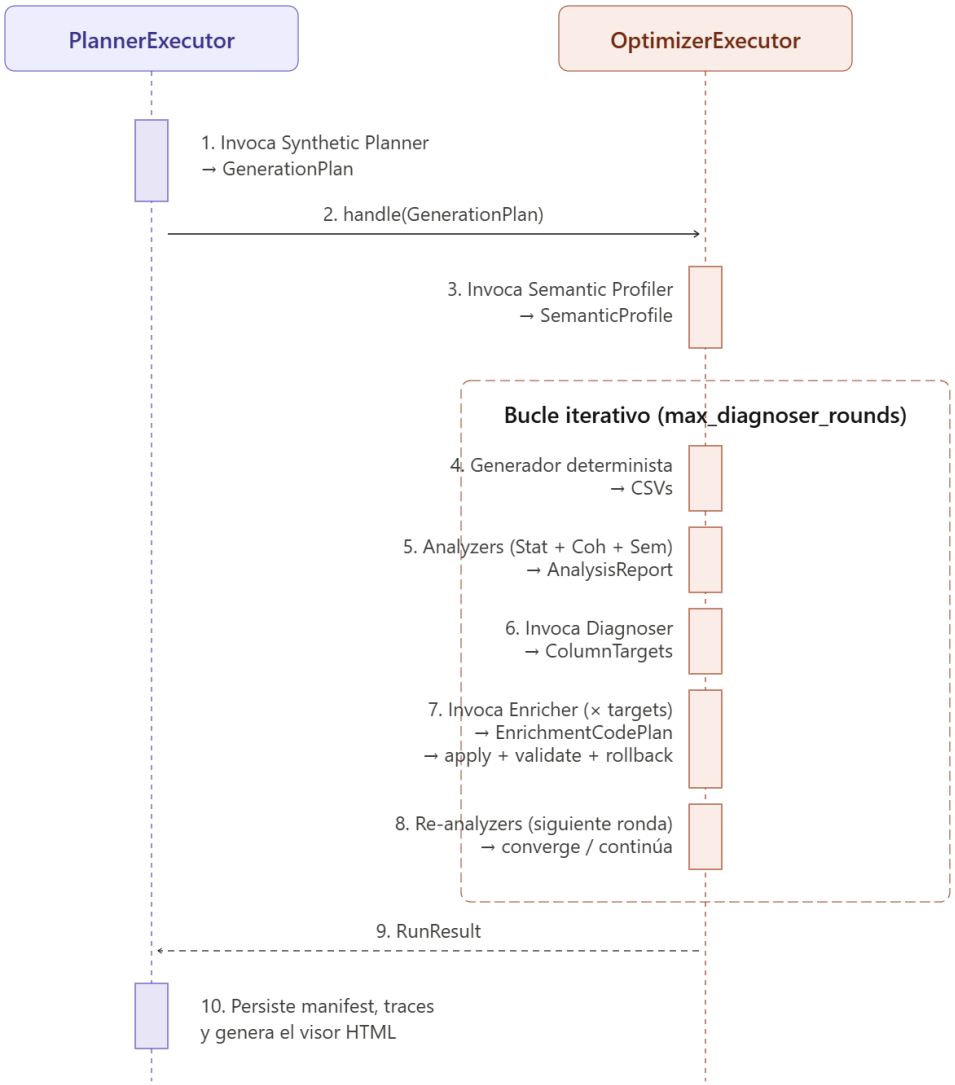

Ilustración 6 — Diagrama de secuencia entre el PlannerExecutor y el OptimizerExecutor

4.2.3 MODELOS DE DATOS

El sistema define un conjunto reducido de tipos de datos que vertebran la comunicación entre capas. DataContractSpec es la representación canónica del contrato parseado: contiene

tablas, columnas, restricciones, relaciones y reglas de calidad, y se mantiene inmutable durante toda la ejecución. GenerationPlan es el contrato entre el Planner y el Optimizer: declara cuántas filas se generan por tabla, qué semilla aleatoria se usa y qué perfil estructural aplica a cada relación. AnalysisReport agrega los hallazgos de los tres analizadores en una estructura común basada en Finding, donde cada finding identifica una columna problemática, su severidad y la evidencia que justifica el problema. ColumnTarget es la unidad de trabajo del Diagnoser: representa una columna concreta (o grupo de columnas) que debe corregirse, con un reason tipificado (fk_alignment, not_null, pattern_shape, semantic_realism, etc.) que guía la posterior decisión del Enricher. Finalmente, EnrichmentCodePlan es el código Python que el Enricher genera para corregir un target concreto, junto con metadatos que permiten aplicarlo de forma trazable y reversible.

4.2.4 DECISIONES ARQUITECTÓNICAS

El diseño descrito incorpora cuatro decisiones técnicas de fondo que merecen justificación explícita, ya que condicionan el resto del sistema y se desvían de aproximaciones más comunes en la literatura.

La primera es la separación entre generador determinista y agentes LLM. Una alternativa frecuente en sistemas similares consiste en delegar la generación completa al modelo [2], pidiéndole que produzca filas de CSV directamente. Esta aproximación es atractiva por su simplicidad pero presenta dos problemas graves: el coste por fila es elevado (cada fila implica tokens de salida) y, sobre todo, no hay garantías estructurales — el modelo puede generar claves foráneas que no existen, violar la cardinalidad de un PK compuesto o ignorar restricciones de patrón. La estrategia adoptada en este trabajo invierte la división del trabajo: el código determinista se encarga de producir datos estructuralmente válidos a costa marginal cero, y el LLM solo interviene cuando se requiere razonamiento semántico (¿este patrón es realista?, ¿estos valores parecen un código de producto?). Esta inversión reduce el coste en al menos un orden de magnitud y elimina por construcción una clase entera de errores.

La segunda es la adopción de un bucle iterativo con rollback en lugar de generación one-shot. Un sistema one-shot genera el dataset una vez y lo entrega; si tiene problemas, se descarta. El sistema desarrollado en este trabajo, en cambio, trata el dataset como un artefacto evolutivo: se genera una primera versión, se analiza, se aplican correcciones quirúrgicas sobre las columnas problemáticas, y se valida que las correcciones no han introducido regresiones. Si las introducen, se revierte la corrección y se intenta otro enfoque. Esta aproximación tiene un coste en complejidad —requiere infraestructura de snapshot y restore sobre los CSV, propagación de cambios a través del grafo de FKs, y detección de regresiones— pero compensa con creces: permite alcanzar niveles de calidad que un sistema one-shot no puede, y proporciona una auditoría completa del proceso correctivo.

La tercera es el uso de OpenTelemetry estándar en lugar de logging propietario. Las herramientas comerciales del ecosistema LLM (Langfuse, Phoenix, LangSmith) ofrecen experiencias muy pulidas pero atan al usuario a su backend específico. OpenTelemetry, en cambio, define un protocolo abierto y un conjunto de convenciones semánticas (las GenAI semantic conventions publicadas en 2024) [9] que cualquier backend puede consumir. El sistema desarrollado emite trazas conformes a esas convenciones, lo que significa que un mismo run puede inspeccionarse con Jaeger, importarse a Tempo, analizarse con un script Python ad-hoc o visualizarse con el visor HTML desarrollado en este trabajo, sin modificación alguna del código.

La cuarta es el uso de salidas estructuradas (structured outputs) en todos los agentes. Un agente que devuelve texto libre obliga al consumidor a parsearlo, lo que introduce fragilidad. Los structured outputs basados en Pydantic obligan al modelo a producir JSON conforme a un esquema declarado, lo que resuelve por construcción los problemas de parseo y permite tipar las respuestas en el resto del código. El coste es un prompt más largo (el modelo recibe el esquema implícitamente) y, ocasionalmente, respuestas que no encajan; el sistema absorbe estos casos con un parser tolerante que primero intenta el camino estructurado y, si falla, busca el primer bloque JSON balanceado en el texto plano de la respuesta.

4.3 IMPLEMENTACIÓN

4.3.1 PARSER Y MODELO DE CONTRATO

El primer eslabón de la canalización es el parser de contratos ODCS, encapsulado en el módulo core.parser. Su responsabilidad es transformar un fichero YAML conforme al Open Data Contract Standard en una representación interna basada en dataclasses de Python. La elección de dataclasses sobre alternativas como Pydantic responde a un criterio funcional: el modelo de contrato es inmutable durante toda la ejecución y no requiere validación dinámica más allá de la inicial; dataclasses ofrece esto con mínima sobrecarga y cero dependencias adicionales.

La traducción del YAML a la representación interna no es directa por dos motivos. El primero es que ODCS admite múltiples sintaxis para el mismo concepto. Las claves foráneas, por ejemplo, pueden declararse en tres formas distintas según el contrato: como una cadena "tabla.columna" (típico en relaciones simples de columna), como una lista de cadenas (para relaciones compuestas a nivel de tabla), o como un diccionario {table, column} (forma no estándar pero presente en algunos toolchains del ecosistema). El parser absorbe estas variaciones mediante la función _coerce_to_ref, que normaliza cualquiera de los tres formatos a una tupla (tabla, columna) consumible por el resto del sistema.

El segundo motivo es la tolerancia a fallos parciales. Los contratos reales de la industria contienen con frecuencia secciones malformadas o no canónicas: una relación sin campo type, un customProperty con estructura inesperada, una columna con physicalName pero sin name. El parser implementa un principio de fail-soft: cuando encuentra un elemento que no puede mapear, emite un warning estructurado al log y continúa con el resto del contrato, en lugar de abortar el parseo completo. Esta política es crítica para la usabilidad del sistema, ya que permite procesar contratos de producción sin requerir que estén perfectamente conformes al estándar.

El modelo de datos resultante se compone de cinco dataclasses principales. DataContractSpec representa el contrato completo y contiene la lista de tablas más metadatos

del header (versión, dominio, data product, propósito). Las secciones del YAML que el parser no mapea explícitamente se preservan bajo raw_sections, lo que permite acceder a ellas desde código posterior sin pérdida de información. TableSpec modela una tabla con sus columnas, sus reglas de calidad a nivel de tabla y sus relaciones de tabla (típicamente claves foráneas compuestas). ColumnSpec agrupa la información de una columna: tipo lógico, tipo físico, sus restricciones (ColumnConstraints), su semántica libre (ColumnSemantics), sus reglas de calidad y sus relaciones. La separación entre ColumnConstraints y ColumnSemantics no es meramente organizativa: la primera contiene la información que el generador determinista necesita (PK, unicidad, dominio, longitudes), mientras que la segunda contiene la información que los agentes LLM consumen (descripción en lenguaje natural, business name, clasificación, tags).

QualityRule merece mención aparte. ODCS expresa restricciones que no encajan en el sistema de tipos —por ejemplo "esta columna no puede contener valores nulos" o "el número de filas debe ser mayor que 50.000"— mediante un bloque quality con una métrica nombrada (nullValues, rowCount, invalidValues) y una comparación. Estas reglas se preservan en el modelo y son consumidas tanto por el generador determinista (la regla nullValues mustBe 0 se interpreta como required, aun cuando el campo required no esté marcado a nivel de columna) como por el validador (que comprueba a posteriori que el dataset cumple cada regla). [1]

4.3.2 GENERADOR DETERMINISTA

El generador determinista, implementado en core.generator, es el componente que transforma un DataContractSpec y un GenerationPlan en filas concretas listas para escribir a CSV. Es el componente más crítico del sistema en términos de garantías estructurales: si el generador falla en respetar una FK, ningún agente posterior podrá repararlo correctamente, ya que las correcciones se aplican como patches sobre el dataset producido.

El diseño del generador parte del análisis topológico del grafo de dependencias entre tablas. Las tablas conectadas por relaciones de FK forman un grafo dirigido cuya estructura determina el orden de generación. Si el grafo es acíclico, las tablas pueden generarse en

orden topológico: cada tabla hija se genera después de sus padres, y las claves foráneas se resuelven muestreando filas ya materializadas del padre. Pero los contratos reales contienen ciclos: una tabla departments que tiene un head_employee_id apuntando a employees, que a su vez tiene un department_id apuntando a departments. Estos casos forman strongly connected components (SCC) en el grafo de dependencias y no admiten un orden topológico trivial. El generador detecta los SCC mediante el algoritmo de Tarjan [15] y los procesa por una vía distinta: la resolución mediante un solucionador de restricciones CP-SAT, implementada en el módulo core.structure_solver.

La rama acíclica funciona fila a fila. Para cada tabla, el generador construye un bucle externo que itera n_rows veces; en cada iteración produce un diccionario con los valores de cada columna. Las columnas se procesan en tres fases: primero las restricciones estructurales (PK, FK), después las restricciones de tipo lógico (UUID, integer con rango, fecha en intervalo, patrón regex), y finalmente las columnas libres (texto, números sin restricción especial). Cada fase aplica una lógica especializada documentada más adelante. Si una fila resultante colisiona con una ya generada en su tupla de PK, se descarta y se reintenta hasta MAX_PK_RETRIES veces; si tras 200 reintentos no se encuentra una tupla única, se omite la fila y se emite un warning por saturación del dominio del PK, lo que indica al usuario que el contrato está pidiendo más filas únicas de las que el espacio de claves admite.

La rama cíclica requiere una aproximación radicalmente distinta. No es posible decidir el valor de head_employee_id sin haber decidido antes qué employee_id existen, y viceversa. La solución adoptada formula el problema como un problema de satisfacción de restricciones: cada FK en el SCC se modela como una variable entera que indexa una fila del padre, las restricciones del contrato (unicidad, no-nulidad, AllDifferent para FKs marcadas como unique) se traducen a restricciones del solver, y se invoca el motor CP-SAT de Google OR-Tools [5] para encontrar una asignación satisfactoria. El resultado del solver es un conjunto de referencias por fila —"la fila 3 de departments apunta a la fila 17 de employees"— que el generador aplica iterativamente hasta que el grafo de referencias queda completamente resuelto. Solo después se materializan los valores concretos de las columnas no estructurales.

Tres tipos de problemas estructurales requieren tratamiento especializado en el generador, ya que su tratamiento produce errores difíciles de diagnosticar. El primero es el patrón PK simple = FK única hacia un padre único (pk_unique_fk_chain): cuando una tabla tiene un PK simple que es a su vez una FK hacia una columna única del padre, una asignación probabilística produce el conocido problema del cumpleaños, perdiendo aproximadamente un 0.3% de filas por colisiones cuando el tamaño del hijo iguala al del padre. El generador detecta este patrón y conmuta a una asignación determinista por índice (la fila i del hijo toma el padre i), eliminando las colisiones por construcción.

El segundo es el PK compuesto enteramente formado por FKs (pk_is_all_fk): por ejemplo, una tabla transfers cuya clave primaria es (rcvr_id, rcvr_country_code) donde ambas columnas son FKs hacia una tabla receivers con PK compuesto. El generador asigna deterministamente las parejas de la misma fila padre, garantizando que la pareja existe en el padre.

El tercero, y más sutil, es el caso en el que una columna participa simultáneamente en una FK simple a nivel de columna y en una FK compuesta a nivel de tabla. Sin tratamiento especial, la FK simple asignaría primero la columna rcvr_id desde un padre cualquiera, y la FK compuesta a nivel de tabla intentaría después asignar la pareja completa, encontrándose con que rcvr_id ya tiene un valor que probablemente no casa con el country_code que la FK compuesta pretendía asignar. El generador resuelve este caso procesando primero las FKs compuestas a nivel de tabla, asignando todas sus columnas atómicamente desde la misma fila padre, y procesando después las FKs simples solo sobre las columnas no claimed por una FK compuesta. Esta sutileza, que en una implementación naïve produce millones de violaciones de FK en datasets grandes, queda resuelta por construcción.

Más allá de las FKs, el generador implementa una resolución cuidadosa de dominios saturables. Una columna PK declarada como varchar(2) con un patrón letras-mayúsculas puede contener como máximo $26^2 = 676$ valores únicos. Si el contrato pide más filas, el sistema lo detecta y produce un error explícito en lugar de generar duplicados silenciosamente. Para columnas con dominios pequeños y enumerables, el generador

conmuta a una enumeración determinista (AAA, AAB, AAC...) que garantiza la cobertura completa del espacio.

Finalmente, el generador respeta la regla de calidad nullValues mustBe 0 interpretándola como required equivalente. Esto resuelve un caso real frecuente en contratos de producción: una columna marcada con required: false a nivel de campo pero con una quality rule que prohíbe nulos. Sin esta interpretación, el generador produciría nulos legítimos según el campo required, que el validador rechazaría inmediatamente por la regla de calidad, creando un ciclo infinito de correcciones.

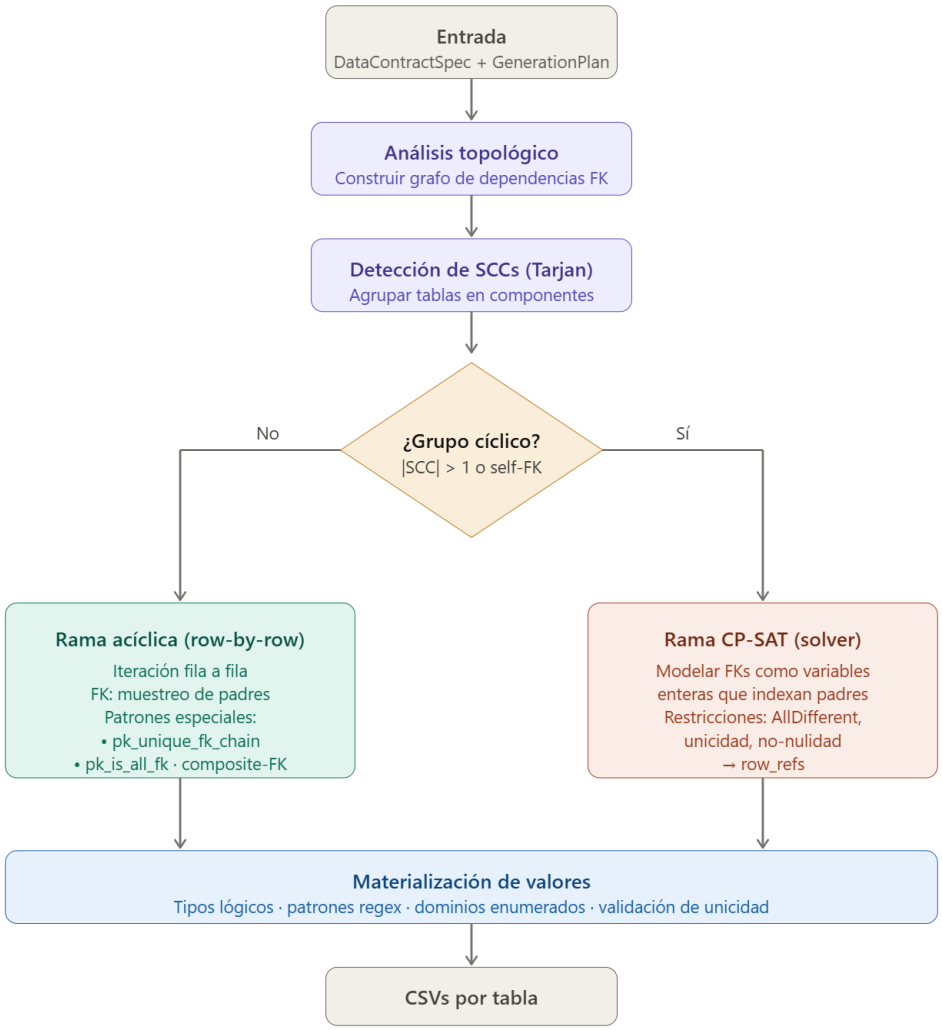

Ilustración 7 — Flujo del generador determinista con sus dos ramas y la fase de materialización

4.3.3 AGENTES LLM

El sistema integra cinco agentes basados en Large Language Models, todos ellos invocando el modelo GPT-4.1-mini a través de la API de Azure OpenAI [8]. La elección del modelo responde a un compromiso entre capacidad de razonamiento y coste operativo: GPT-4.1-mini ofrece structured outputs nativos, ventana de contexto de 128K tokens y un coste aproximadamente diez veces inferior a los modelos de gama alta, lo que permite que un run completo sobre un contrato realista cueste céntimos en lugar de euros.

Cada agente está implementado como una instancia de ChatAgent del Microsoft Agent Framework [7], configurada mediante un fichero YAML que declara su system prompt, su esquema de salida estructurada (clase Pydantic) y sus parámetros de generación (temperatura, top-p, max tokens). Esta separación entre código y configuración permite ajustar el comportamiento de un agente —endurecer un criterio, añadir un ejemplo few-shot, modificar el formato de salida— sin tocar la lógica del workflow.

Los cinco agentes se reparten responsabilidades claramente diferenciadas:

- Synthetic Planner: traduce un contrato ODCS en un GenerationPlan. Decide cuántas filas generar por tabla respetando las reglas de calidad declaradas (típicamente rowCount), asigna semillas aleatorias y selecciona un perfil estructural por relación.
- Semantic Profiler: examina el contrato y produce un SemanticProfile con expectativas semánticas: qué columnas deberían correlacionar, qué distribuciones son plausibles, qué invariantes de agregación deberían cumplirse. Se invoca una sola vez por run, antes de la primera ronda del Diagnoser.
- Semantic Analyzer: examina muestras del dataset generado y detecta problemas que las heurísticas estadísticas no pueden capturar (texto sin sentido, mismatches de dominio, formatos incoherentes con la descripción del contrato).
- Dataset Diagnoser: convierte los hallazgos del análisis en targets accionables (ColumnTarget), priorizando los problemas estructurales sobre los problemas de calidad puramente semánticos.

- Synthetic Enricher: para cada target, genera código Python que corrige la columna afectada y devuelve un EnrichmentCodePlan ejecutable.

Los tres primeros agentes se invocan en una secuencia lineal por cada ronda del bucle principal, mientras que el Diagnoser y el Enricher operan sobre los resultados acumulados. A continuación, se describe cada uno con detalle.

4.3.3.1 Synthetic Planner

El Planner recibe el contrato YAML completo y, opcionalmente, un bloque de planning_feedback cuando se invoca en modo retry tras un fallo del generador. Su salida estructurada es un GenerationPlan con tres campos principales: rows_per_table (un diccionario tabla → entero), default_rows (valor por defecto para tablas no mencionadas explícitamente) y structure_profile (un objeto que parametriza el comportamiento del generador en relaciones específicas, por ejemplo declarando que una FK es de tipo one-to-one o one-to-many sesgada).

La complejidad del Planner no está en el código que lo invoca —apenas treinta líneas en runners.py— sino en su system prompt, que codifica el conocimiento experto sobre cómo dimensionar un dataset coherente. El prompt contiene reglas explícitas sobre cómo respetar rowCount mustBeGreaterThan, cómo manejar tablas hijas con FKs hacia padres más pequeños (la tabla hija debe escalarse hacia abajo, no el padre hacia arriba), y cómo reaccionar a un feedback del generador que reporta una infeasibilidad. Este último punto es crítico: cuando el generador falla con un error como not enough unique parent values, el Planner recibe el mensaje de error junto con el plan anterior y debe producir un plan revisado que ataque la causa raíz, típicamente reduciendo el tamaño de la tabla problemática o ampliando el dominio del padre.

4.3.3.2 Semantic Profiler

El Profiler es el único agente que no consume datos generados, sino solo el contrato. Su tarea es predecir, a partir de la semántica declarada por el contrato (descripciones, tipos lógicos,

business names, tags), qué propiedades semánticas deberían tener los datos sintéticos para considerarse realistas. Su salida es un SemanticProfile con cuatro secciones:

- expected_correlations: pares de columnas que deberían estar correlacionadas (por ejemplo, salary y grade en una tabla de empleados).
- expected_ordering: columnas con orden esperado (fechas que deben ser monótonas, grades con jerarquía).
- expected_distributions: forma de la distribución esperada por columna (pareto_long_tail, gaussian, uniform_ok).
- expected_invariants: relaciones de agregación entre tablas (por ejemplo, la suma del campo amount en transactions debe coincidir con el campo total correspondiente en accounts).

El Profiler se invoca una única vez por run, en la primera ronda del Diagnoser, y su resultado se cachea para todas las rondas posteriores. Esto evita reanalizar la semántica del contrato en cada iteración, ahorrando tokens significativamente.

4.3.3.3 Capa de análisis: Profiler, Statistical, Coherence y Semantic Analyzer

La capa de análisis está implementada en core.analysis y orquesta cuatro analizadores complementarios que cooperan para construir el AnalysisReport. El primero, el Semantic Profiler, se invoca una sola vez por run antes de la primera ronda y produce el SemanticProfile que el resto de la capa consume; los otros tres se ejecutan en cada ronda del bucle iterativo y operan sobre el dataset materializado.

El primer analizador es el Semantic Profiler, descrito en detalle en la subsección anterior. Es un agente LLM que examina exclusivamente el contrato (no consume datos) y produce las expectativas semánticas que los analizadores posteriores utilizan como hipótesis a contrastar: correlaciones esperadas entre columnas, invariantes de agregación entre tablas, distribuciones plausibles y órdenes naturales. Su rol es por tanto el de fuente de verdad semántica que el resto de la capa intenta verificar.

Los dos siguientes son completamente deterministas y no consumen LLM:

- Statistical Analyzer: calcula estadísticos descriptivos por columna (cardinalidad, unique rate, null rate, mínimo, máximo, skewness) y detecta anomalías como columnas con unique rate anómalo respecto a su tipo declarado.
- Coherence Analyzer: aplica las reglas declaradas en el SemanticProfile (correlaciones esperadas, invariantes de agregación, secuencias temporales) sobre los datos reales y reporta divergencias entre lo que el contrato declara como esperable y lo que los datos generados muestran.

El cuarto analizador, el Semantic Analyzer, es el segundo agente LLM de la capa. Su responsabilidad es detectar problemas que ni la estadística ni las reglas declaradas pueden capturar: una columna product_code que el generador rellena con "qx7m", "abkt", "mhwz" —cadenas válidas según la longitud del campo pero claramente irreales—, o una columna customer_name con valores "AAAAA", "BBBBB", "CCCCC" que satisfacen el dominio de letras pero no parecen nombres.

El analizador construye una lista de candidatos —una entrada por columna con una cantidad reducida de muestras (20 valores) y los metadatos del contrato— y se la envía al LLM con instrucciones explícitas: clasificar cada columna en uno de tres buckets (identificador/código, texto humano, numérico/temporal/otro) y reportar problemas semánticos en un SemanticFindingsReport estructurado.

Un detalle de implementación importante es el procesamiento por lotes paralelos. Inicialmente el analizador enviaba todos los candidatos en un único prompt, pero esto producía dos problemas: prompts excesivamente largos que degradaban la calidad de las respuestas (el conocido fenómeno lost in the middle [6]), y latencias elevadas. La versión actual divide los candidatos por tabla y los procesa en lotes de tres tablas en paralelo mediante asyncio.gather, con manejo independiente de fallos: un lote que falla es registrado pero no aborta la ejecución del resto.

4.3.3.4 Dataset Diagnoser

El Diagnoser recibe el AnalysisReport consolidado y produce un TargetReport con la lista de ColumnTarget que el sistema debe corregir. Cada target declara la tabla afectada, las

columnas concretas, una razón tipificada (fk_alignment, not_null, pattern_shape, uniqueness, range_shape, inter_table_coherence, cross_column_coherence, statistical_shape, semantic_realism, temporal_sequence, aggregation_invariant) y un bloque de evidencia que justifica por qué se identifica el problema.

La razón es crítica para el comportamiento del bucle: el OptimizerExecutor ordena los targets por prioridad mediante un mapa fijo (_REASON_ORDER) que asegura que primero se atacan los problemas estructurales (validación del contrato, alineación de FKs) y solo después los problemas de calidad semántica. Esta priorización es necesaria porque corregir un problema semántico sobre datos estructuralmente inválidos es una pérdida de tokens: si una columna tiene FKs rotas, mejorar su realismo semántico es prematuro.

Como el Semantic Analyzer, el Diagnoser opera en lotes paralelos de tablas, agrupadas por diagnoser_batch_size (configurable, valor por defecto 2). Cada lote produce un TargetReport parcial; el OptimizerExecutor los consolida, aplica una barrera de seguridad sobre composiciones de PKs (los targets que tocan parcialmente una clave compuesta son normalizados o bloqueados) y deduplica por (tabla, columnas, razón).

4.3.3.5 Synthetic Enricher

El Enricher es el agente más exigente del sistema, ya que su salida no es texto descriptivo ni un objeto de configuración, sino código Python ejecutable. Para cada ColumnTarget recibe el contrato, el plan de generación, el reporte del Diagnoser filtrado a la tabla relevante, una muestra de los datos actuales, el invalid feedback de intentos anteriores (cuando los hay) y debe devolver un EnrichmentCodePlan con un fragmento de código que, ejecutado sobre el CSV objetivo, corrija la columna afectada.

El esquema de salida EnrichmentCodePlan impone una estructura rígida al código generado: una función con firma fija que recibe un pandas.DataFrame y devuelve el DataFrame modificado. El runtime del apply (_apply_enrichment_codegen_plan_ordered en runners.py) compila ese código en un sandbox con los imports necesarios y lo ejecuta sobre el CSV en disco.

El Enricher opera en dos modos: en el primer intento de cada target trabaja a partir del contrato y los datos actuales; si su código falla en la validación posterior, se le invoca de nuevo con el invalid_feedback —los errores concretos detectados— para que produzca una versión revisada. Tras tres intentos fallidos sobre el mismo target, el sistema lo da por perdido y revierte cualquier cambio aplicado.

Una optimización relevante es el prefetch de planes. Cuando el Optimizer tiene la lista completa de targets válidos de una ronda, lanza en paralelo (vía asyncio.gather con un semáforo de concurrencia) la generación del primer plan de cada target, mientras simultáneamente comienza a aplicar los planes ya recibidos sobre los CSV. Esto solapa la latencia del LLM con el trabajo de aplicación, reduciendo el tiempo total de ronda en aproximadamente un 30-40%.

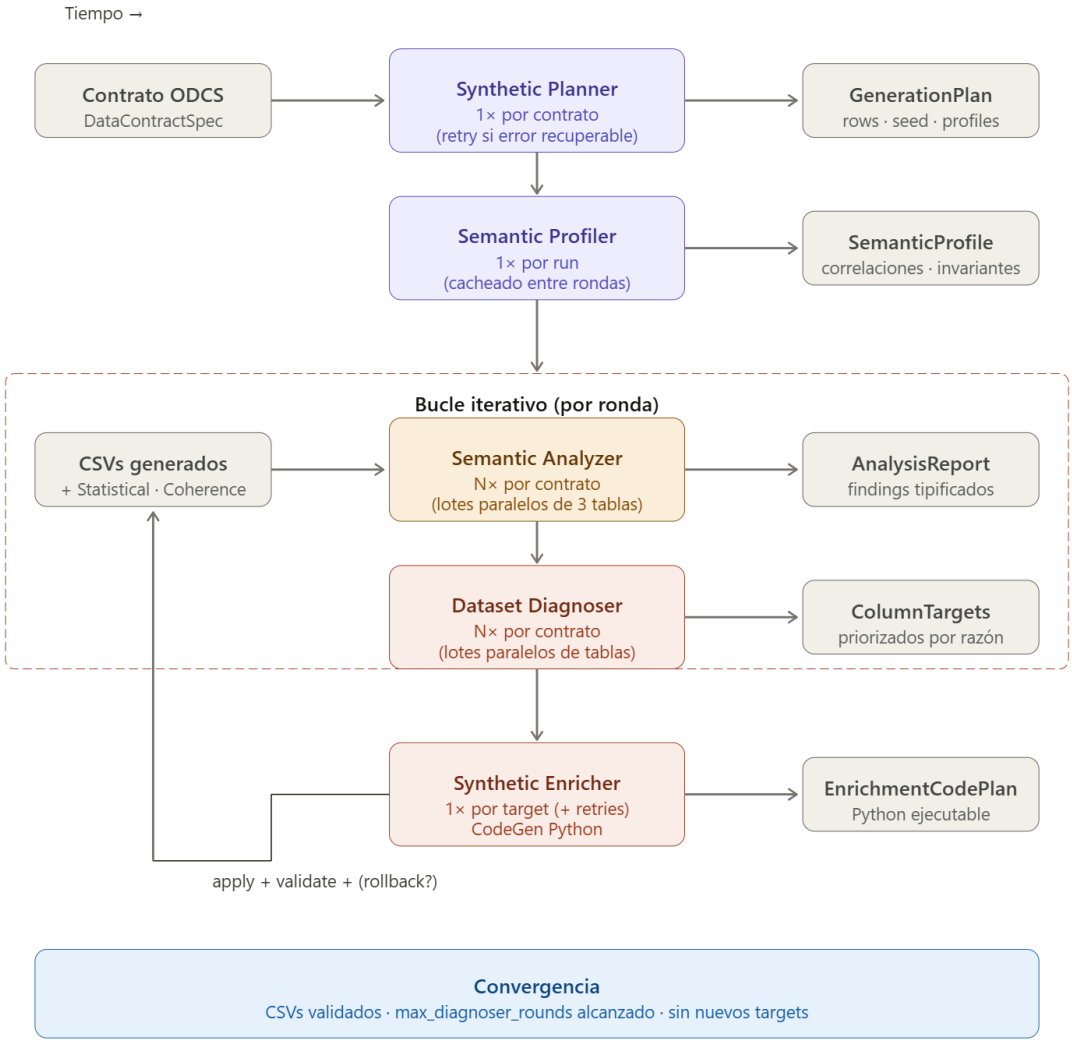

Ilustración 8 — Los cinco agentes LLM con sus inputs, outputs y orden temporal.

4.3.4 BUCLE ITERATIVO DE AUTO-CORRECCIÓN

Las secciones anteriores han descrito los componentes del sistema de forma aislada: el generador determinista, los cinco agentes LLM y la capa de análisis. Esta sección describe cómo se orquestan en una ejecución completa y, sobre todo, cómo el sistema reacciona ante fallos parciales para producir un dataset de calidad creciente. Toda la lógica de orquestación reside en el método _process_single_contract del OptimizerExecutor, que implementa la máquina de estados conceptual que se describe a continuación.

4.3.4.1 Diagrama de estados conceptual

Una ejecución sobre un contrato dado atraviesa cuatro fases. La primera es la fase de generación, en la que el generador determinista intenta materializar el GenerationPlan recibido. Esta fase puede tener éxito —en cuyo caso se entra en la fase de análisis— o fallar con un error que el sistema clasifica como recuperable o fatal. Si el error es recuperable (por ejemplo, not enough unique parent values o domain exhausted), el sistema invoca al Planner en modo retry con el error como feedback, recibe un nuevo plan y reintenta la generación. Este bucle de retry del Planner tiene un presupuesto fijo (max_planner_retries = 3); si se agota sin éxito, el contrato se marca como fallido y el bucle termina.

La segunda fase es la fase de análisis. Tras una generación exitosa, los tres analizadores (estadístico, coherencia, semántico) examinan el dataset y producen un AnalysisReport. Si el análisis no detecta hallazgos accionables, el contrato se da por completado y el bucle termina con éxito. Si los detecta, se entra en la fase de diagnóstico.

La tercera fase es el diagnóstico y enriquecimiento por rondas. El Diagnoser convierte los hallazgos en ColumnTarget priorizados, y el Enricher los procesa secuencialmente con apply-validate-rollback. Tras procesar todos los targets válidos de una ronda, el sistema vuelve a entrar en la fase de análisis para verificar el estado del dataset. Cada iteración constituye una ronda; el bucle ejecuta hasta max_diagnoser_rounds = 3 rondas o termina antes si se cumple alguna de las condiciones de parada: el Diagnoser no produce nuevos targets, todos los targets candidatos ya han sido procesados o todos los targets tocan exclusivamente columnas ya corregidas.

La cuarta fase es la finalización, en la que el sistema consolida los artefactos del run (datos generados, manifest, trazas) y produce el resumen final con las métricas de éxito.

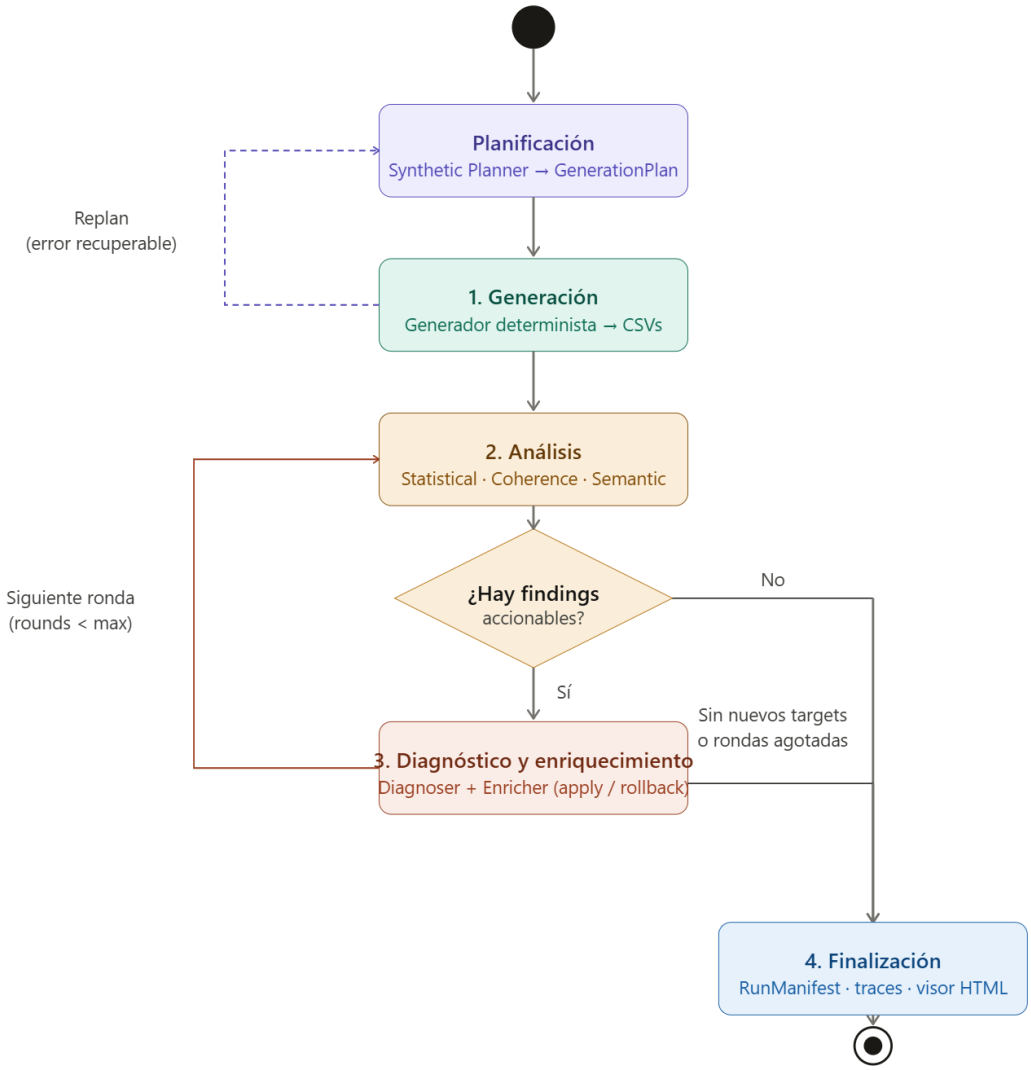

Ilustración 9 — Diagrama de estados del bucle iterativo de auto-corrección.

4.3.4.2 Política de rollback

El mecanismo de rollback es la pieza que permite tratar el dataset como un artefacto evolutivo sin riesgo de degradación. El principio rector es simple: un cambio que empeora el dataset debe revertirse. La complejidad está en definir operativamente qué significa "empeorar".

La implementación adopta un enfoque basado en fingerprints de issues. Antes de procesar un target, el sistema toma una snapshot de las columnas afectadas y de las columnas

dependientes de FKs (es decir, las columnas hijas que apuntan al target en otras tablas). A continuación, registra el conjunto de issues presentes en el validation_report.json antes del cambio, computa una fingerprint canónica para cada issue —típicamente la tupla (tabla, columna, regla, ejemplo) serializada— y guarda el conjunto de fingerprints.

Tras aplicar el plan del Enricher y revalidar, el sistema vuelve a calcular el conjunto de fingerprints. La diferencia entre el conjunto post-apply y el conjunto pre-apply identifica los issues introducidos por el cambio. Si esta diferencia no está vacía, el sistema considera que el cambio ha introducido una regresión y dispara el rollback: restaura las columnas desde la snapshot, restaura las columnas dependientes, y revalida para confirmar que el estado pre-apply ha sido recuperado correctamente.

Esta política tiene una propiedad importante: el rollback no se dispara solo por la persistencia de los issues originales, sino exclusivamente por la aparición de nuevos. Esto es deliberado: un Enricher puede no conseguir resolver completamente el problema que se le ha asignado, y eso no es razón para revertir su trabajo si los issues que persisten son los mismos que ya estaban antes. Solo se revierte cuando el Enricher ha empeorado activamente el dataset.

El rollback se implementa mediante el módulo contract_helpers, que ofrece dos primitivas básicas: _snapshot_columns_with_locator y _restore_columns_with_locator. Ambas operan a nivel de fila identificando las filas por su PK (no por índice posicional, ya que los enrichers pueden reordenar el CSV). El "locator" es una función que mapea PKs a posiciones en el DataFrame y permite restaurar valores aunque el orden físico del fichero haya cambiado entre el snapshot y la restauración.

4.3.4.3 Política de retry del Planner

Cuando el generador falla, no todos los errores son recuperables. La función _is_replannable_generation_error clasifica el error en función de su mensaje, comparándolo con un conjunto de patrones reconocidos:

- table-level fk to ... is empty
- not enough unique parent values

- impossible to generate ... rows
- domain exhausted
- unable to generate a unique row
- value space is only ...

Estos patrones identifican infeasibilidades estructurales que un nuevo plan puede resolver: típicamente reduciendo el tamaño de la tabla problemática, reescalando el padre o cambiando la cardinalidad declarada. Si el error no encaja con ninguno de estos patrones, se considera fatal y la excepción se propaga sin más intervención. Esta clasificación conservadora es deliberada: un retry sobre un error que el Planner no puede resolver solo consume tokens y latencia sin posibilidad de éxito.

Cuando el retry procede, el sistema construye un objeto de planning_feedback que incluye el plan anterior, el mensaje de error completo, las tablas involucradas (extraídas mediante una expresión regular que captura nombres entrecomillados del mensaje) y un objetivo explícito (repair_goal) que orienta al Planner sobre la naturaleza del fallo. Este objeto se inyecta en el system prompt del Planner como contexto adicional y la nueva invocación produce un plan revisado.

4.3.4.4 Pre-injectors deterministas

Una optimización significativa del bucle es la pre-inyección de targets deterministas. Antes de invocar al Diagnoser para que analice los hallazgos, el sistema procesa el SemanticProfile con dos pre-injectors implementados en pre_injectors.py:

- inject_invariant_targets: convierte cada invariante de agregación declarado por el Profiler (por ejemplo, "el campo total de la tabla padre debe ser igual a la suma del campo amount de la tabla hija agrupada por la FK") en un ColumnTarget con razón aggregation_invariant.
- inject_correlation_targets: hace lo análogo con las correlaciones esperadas, generando targets con razón cross_column_coherence para cada par de columnas que

el Profiler declara correlacionadas y que no presentan correlación observable en los datos.

Estos pre-injectors son completamente deterministas y se ejecutan en milisegundos. Su valor es doble: por un lado, descargan al Diagnoser de tener que redescubrir vía LLM lo que ya está declarado en el contrato (ahorro de tokens); por otro, capturan las columnas reclamadas (claimed_columns) por estos targets deterministas y las pasan al Diagnoser como contexto, indicándole explícitamente que no proponga targets alternativos sobre esas columnas en el mismo round.

El Optimizer aplica un filtro adicional sobre los targets del Diagnoser: cualquier target con razón statistical_shape o semantic_realism que toque una columna ya claimed por un pre-injector es descartado. Esto garantiza que las correcciones deterministas se aplican primero (vía el Enricher sobre los targets pre-inyectados) y los targets puramente semánticos no compiten con ellas.

4.3.4.5 Concurrencia y paralelismo

El bucle hace un uso cuidadoso de la concurrencia para reducir la latencia sin saturar la API del LLM. Se utilizan tres semáforos asyncio.Semaphore que acotan la paralelización en tres puntos clave:

- El Diagnoser procesa lotes de tablas en paralelo (diagnoser_parallelism = 4), de forma que con un batch size de 2 tablas, hasta 4 lotes pueden estar en vuelo simultáneamente.
- El prefetch del Enricher lanza la generación de planes en paralelo (enricher_parallelism = 4), solapando la latencia de generación con el tiempo de aplicación.
- El análisis semántico procesa lotes de tres tablas en paralelo dentro de su propio runner.

En contraste, la aplicación de planes sobre los CSV es estrictamente secuencial, target por target. Esto es deliberado: dos targets que toquen la misma tabla podrían producir

condiciones de carrera sobre el fichero CSV, y el coste de implementar un sistema de bloqueo a nivel de fila no compensa el limitado paralelismo que se ganaría (la mayor parte de la latencia ya está en las llamadas al LLM, no en la I/O de CSV).

El trabajo síncrono que sí se beneficia del paralelismo de E/S —lecturas de CSV, escrituras, snapshots— se ejecuta con asyncio.to_thread, lo que mantiene el event loop libre para servir las llamadas concurrentes al LLM.

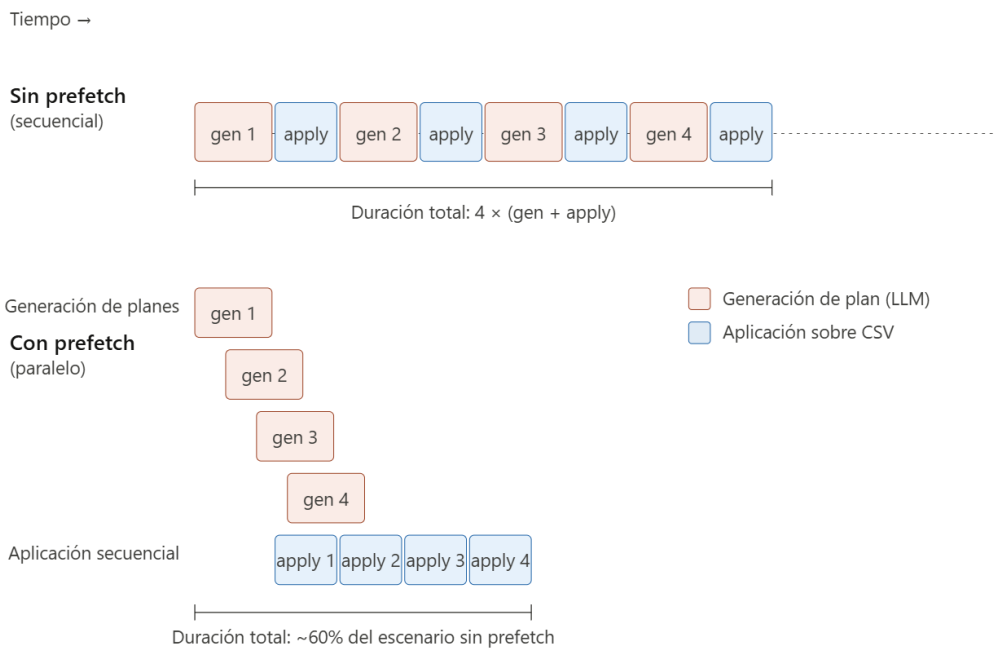

Ilustración 10 — Solapamiento entre la generación de planes (prefetch) y la aplicación secuencial sobre los CSVs.

4.3.5 OBSERVABILIDAD

La observabilidad del sistema es uno de los requisitos no funcionales críticos (RNF2) y condiciona varias decisiones de diseño descritas en secciones anteriores. El objetivo no es simplemente "registrar logs", sino producir un artefacto autocontenido por ejecución que permita reconstruir a posteriori qué ha hecho el sistema, cuántos recursos ha consumido y

por qué ha tomado las decisiones que ha tomado. Para alcanzar este objetivo se ha adoptado OpenTelemetry como estándar técnico, implementando los tres pilares canónicos —trazas, eventos correlacionados y métricas— sobre la infraestructura del Microsoft Agent Framework.

4.3.5.1 Coexistencia con MAF

El Microsoft Agent Framework registra su propio TracerProvider durante la inicialización del workflow, y emite por su cuenta una jerarquía rica de spans bajo las GenAI Semantic Conventions [9] publicadas por la OpenTelemetry Foundation: invoke_agent <nombre> para cada invocación de agente, chat <modelo> para cada llamada al LLM, atributos gen_ai.usage.input_tokens y gen_ai.usage.output_tokens con los conteos de tokens, gen_ai.agent.name, gen_ai.request.model, gen_ai.input.messages, gen_ai.output.messages y otros más. La existencia de este tracing nativo plantea una pregunta de diseño: ¿debe el sistema sustituir el TracerProvider de MAF por uno propio, o debe coexistir con él?

La decisión adoptada es coexistir, por dos razones. La primera es que sustituir el provider de MAF supondría perder los spans gen_ai.*, que son enormemente valiosos para el análisis y siguen un estándar abierto. La segunda es que la coexistencia se consigue mediante un patrón soportado por la API de OTel: en lugar de llamar a trace.set_tracer_provider() (que sobrescribe el provider existente), el código del sistema detecta si ya hay un provider real registrado y, en ese caso, simplemente adjunta sus propios processors al provider existente.

El resultado es que los spans emitidos por MAF y los spans emitidos por el código del sistema (synth.workflow.run) viven bajo el mismo trace_id, anidados correctamente en una única jerarquía, y son procesados por los mismos exporters. La separación entre "spans nuestros" y "spans de MAF" se hace por convención de nombres: los nuestros tienen el prefijo synth.*.

4.3.5.2 Trazas: el primer pilar

El sistema emite un span raíz por cada ejecución del workflow, llamado synth.workflow.run. Este span se abre al inicio del OptimizerExecutor.handle() y se cierra cuando todos los

contratos han sido procesados o el workflow termina por error. Sus atributos resumen el resultado del run —synth.run_id, synth.contracts_count, synth.summary.total_contracts, synth.summary.successful_contracts, synth.summary.total_targets, synth.summary.failed_targets, synth.final_ok— de forma que el span por sí solo cuenta la historia del run sin necesidad de leer el workflow_run_*.json.

Bajo este span raíz, MAF anida automáticamente toda su jerarquía de invoke_agent y chat. El sistema, por tanto, no necesita instrumentar manualmente cada agente ni cada llamada al LLM: basta con asegurar que el código que invoca a los agentes se ejecuta dentro del contexto del span synth.workflow.run.

Los spans se exportan a dos destinos. El primero es un fichero JSONL (<run_dir>/traces.jsonl) escrito por una clase custom FileSpanExporter que serializa cada span mediante el método to_json() del SDK y lo añade como una línea independiente. El formato JSONL es compatible con cualquier herramienta OTel que acepte OTLP-JSON: Jaeger, Tempo, otel-desktop-viewer o un script Python ad-hoc pueden consumirlo sin modificación. El segundo destino es opcional: cuando la variable de entorno SYNTH_OTEL_CONSOLE está activa, los spans se imprimen también por consola mediante el ConsoleSpanExporter estándar del SDK, lo que facilita el desarrollo local. Por defecto el flag está desactivado, ya que el ruido en consola interfiere con los logs estructurados que el sistema emite por separado.

Cada run escribe sus trazas en una carpeta independiente cuyo nombre sigue el patrón YYYY-MM-DD_HH-MM-SS__contrato__run_id, donde run_id es un identificador opaco de 12 caracteres hexadecimales generado al inicio del run. Esta convención permite localizar rápidamente un run concreto por filesystem y, al mismo tiempo, garantiza la unicidad incluso si dos runs se lanzan en el mismo segundo.

4.3.5.3 Eventos correlacionados: el segundo pilar

El sistema emite logs estructurados en cada transición relevante del workflow mediante una función helper _tlog(stage, **kwargs) definida en helpers.py. Originalmente esta función simplemente imprimía a stdout con un formato fijo:

[WF 14:23:11] generate_start contract=orders.odcs.yaml rows={'tbl': 1000001}

Este formato es legible para humanos pero inútil para análisis posterior: no hay correlación con las trazas, no hay forma de filtrar por run y la línea de log no contiene identificadores de la ejecución. Para resolver este problema sin introducir nuevas dependencias, se adoptó la decisión de adjuntar OTel Span Events al span activo en cada llamada de _tlog.

Los Span Events [10] son una característica nativa de la especificación OTel: cada span puede contener una lista de eventos con timestamp y atributos. Estos eventos se serializan junto con el span en el traces.jsonl, viven bajo el mismo trace_id y span_id que el span que los contiene, y son visibles en cualquier herramienta OTel estándar.

La decisión de usar Span Events en lugar del OTel Logs SDK —que también habría resuelto el problema de la correlación— se tomó por tres motivos. El primero es la ausencia de dependencias adicionales: el Logs SDK requiere su propio LoggerProvider, su propio exporter y configuración de protocolo, mientras que los Span Events están disponibles de forma nativa en cualquier código que ya emita trazas. El segundo es la simplicidad operativa: con un único fichero JSONL por run se obtienen trazas, eventos y métricas a la vez, reduciendo la superficie de fallo. El tercero es la fidelidad de la correlación: un Span Event hereda automáticamente el trace_id y span_id del span activo en el momento de su emisión, sin riesgo de descorrelación por relojes desincronizados o por procesos paralelos.

El resultado es que cada _tlog produce dos efectos. Por un lado, sigue imprimiendo la línea legible por humanos en stdout —un canal útil durante el desarrollo. Por otro, deja un evento estructurado en el span correspondiente del traces.jsonl, navegable por cualquier herramienta OTel.

Ilustración 11 — Detalle de un span en el visor HTML con sus Span Events correlacionados.

4.3.5.4 Métricas: el tercer pilar

El sistema implementó originalmente un MeterProvider propio con un FileMetricExporter custom que serializaba snapshots de métricas a un fichero metrics.jsonl paralelo al traces.jsonl. Las métricas registradas eran cinco: synth.run.duration (histograma), synth.contracts.processed, synth.contracts.failed, synth.targets.processed y synth.targets.failed (contadores).

Sin embargo, durante la fase de pruebas se identificó un problema operativo de timing: el PeriodicExportingMetricReader del SDK no garantiza que un force_flush() bloquee hasta que el exporter custom haya terminado de escribir, lo que provocaba ocasionalmente que el fichero metrics.jsonl se leyera vacío inmediatamente después del shutdown del provider. Tras varios intentos de mitigar este problema mediante os.fsync() y otros mecanismos del sistema operativo, se tomó la decisión de eliminar el fichero metrics.jsonl y derivar las

métricas a posteriori desde los datos que ya están garantizados en disco: las trazas y el manifiesto.

Esta decisión es defendible por dos razones. La primera es que las métricas relevantes —duración total, número de contratos exitosos, número de targets procesados— se pueden recomputar con coste cero a partir del span synth.workflow.run y del run_manifest.json, ambos disponibles antes de la generación del visor. La segunda es que MAF ya emite por su cuenta dos métricas extremadamente útiles que no se pueden derivar trivialmente: gen_ai.client.token.usage (con dimensiones por tipo de token y por modelo) y gen_ai.client.operation.duration (con dimensiones por operación). Estas métricas siguen el estándar OTel GenAI y, cuando están presentes en metrics.jsonl, son consumidas por el visor; cuando no lo están, el visor las calcula desde los spans (sumando los gen_ai.usage.*_tokens de cada span chat).

Esta arquitectura de "métricas con fallback derivado" tiene la propiedad deseable de que el visor funciona correctamente tanto con como sin el fichero de métricas, lo que aumenta la robustez del sistema frente a fallos de timing en el exporter.

4.3.5.5 El RunManifest

Más allá de los tres pilares OTel, el sistema escribe un artefacto adicional por run llamado run_manifest.json. Este fichero no es estándar OTel: es una estructura JSON ad-hoc que resume la ejecución de forma legible por humanos sin necesidad de procesar el traces.jsonl. Contiene el run_id, las rutas de los contratos procesados, el resultado por contrato (ok, out_dir, tables), el resumen agregado (total_contracts, successful_contracts, failed_contracts, total_targets, failed_targets), las marcas temporales de inicio y fin, y un flag crashed que indica si el run terminó por excepción.

El manifest se finaliza y escribe en el bloque finally del OptimizerExecutor, antes del shutdown() de la observabilidad. Este orden es deliberado: si la escritura del manifest fallara, no debe impedir que el BatchSpanProcessor haga flush de los spans pendientes a disco. Y si el workflow lanza una excepción, el manifest se escribe con crashed=True antes de que la

excepción se propague, garantizando que un análisis post-mortem dispone siempre de un resumen estructurado.

4.3.5.6 Visor HTML autocontenido

El último componente de la capa de observabilidad es un visor HTML standalone generado al final de cada run. Su existencia responde a una observación práctica: aunque las trazas en traces.jsonl son consumibles por herramientas OTel estándar, el flujo "abrir Jaeger, importar el fichero, navegar la traza" es demasiado fricción para un análisis rápido. El visor HTML permite abrir la traza con un doble click sobre el index.html del directorio del run, sin servidor, sin instalación, sin credenciales.

El visor está implementado en observability.viewer como un único módulo Python. Su entrada es un directorio de run (lee traces.jsonl, metrics.jsonl si existe, y run_manifest.json); su salida es un fichero index.html con CSS y JavaScript embebidos que se autocompila visualmente al abrirse en el navegador. La generación se invoca automáticamente desde el OptimizerExecutor después del shutdown de la observabilidad, y manualmente mediante python -m observability.viewer <run_dir> sobre cualquier run histórico.

El visor presenta cuatro secciones. La cabecera muestra el run_id, la duración total, el estado final (OK/FAILED/CRASHED), el número de contratos exitosos y fallidos, los targets procesados, los tokens consumidos por dirección (entrada/salida) y una tabla agregada de tokens por agente. La sección detalle por contrato lista los contratos del run con su resultado individual y el número de tablas generadas. La sección timeline representa los spans como un Gantt jerárquico con colores por categoría: verde para los spans del workflow propio (synth.*), morado para los invoke_agent de MAF, azul para los chat <modelo> que representan llamadas concretas al LLM, gris para los executors y otros spans de infraestructura. Cada barra es clicable y abre el panel detalle de span a la derecha, que muestra los identificadores del span (span_id, parent_id, trace_id), todos sus atributos y los Span Events anidados con su offset relativo desde el inicio del span.

Por defecto, las barras correspondientes a los spans chat <modelo> están ocultas para reducir el ruido visual: en la práctica duran lo mismo que su invoke_agent padre, ya que el agente

apenas hace preprocesamiento alrededor de la llamada al LLM. Un botón "Show LLM spans" permite expandirlas cuando el usuario quiere inspeccionar las llamadas individuales.

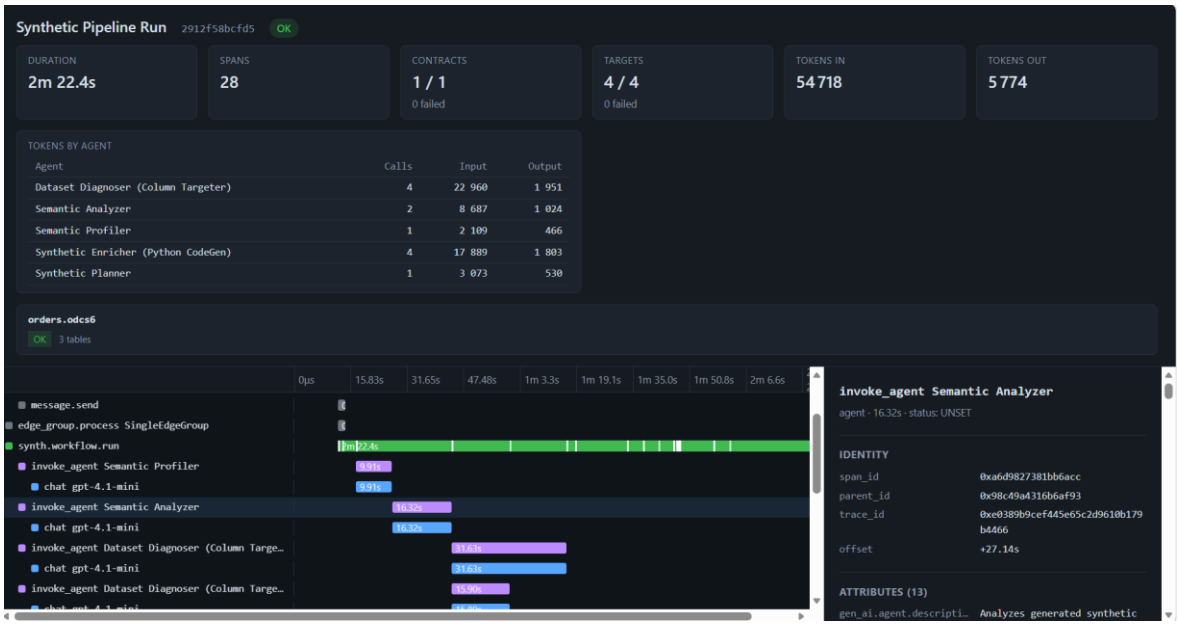

Ilustración 12 — Visor HTML con un run completo del sistema mostrando los LLM spans.

Capítulo 5. ANÁLISIS DE RESULTADOS

Este capítulo presenta la evaluación cuantitativa del sistema desarrollado. La evaluación se ha realizado sobre una batería de diez contratos ODCS que cubren un espectro amplio de complejidad estructural y semántica, ejecutados en una única sesión de pruebas consolidada. Esta forma de evaluación —procesar los diez contratos en un mismo run del workflow— permite obtener métricas agregadas comparables y, al mismo tiempo, observar la robustez del sistema cuando opera de forma sostenida sobre cargas heterogéneas.

5.1 DISEÑO DEL EXPERIMENTO

5.1.1 CONTRATOS DE EVALUACIÓN

La batería de prueba se ha construido con el objetivo de estresar capacidades específicas del sistema mediante contratos sintéticos diseñados ad-hoc, complementados con contratos realistas tomados de repositorios públicos. La tabla siguiente resume el caso ejercitado por cada contrato.

Contrato	Tablas	Característica principal
orders	2	Caso base con FK simple y quality rules clásicas
orders.odcs2	2	FK compuesta a nivel de tabla, PK simple = FK a padre con PK compuesto, escala de 1M filas
orders.odcs3	68	Contrato AdventureWorks de Microsoft con FKs

		implícitas, tipos físicos PostgreSQL nativos
orders.odcs4	4	PKs y FKs compuestas, dominios pequeños con patrón regex
orders.odcs5	4	Bridge table con PK compuesta de tres columnas, dominio bounded único
orders.odcs6	3	Self-reference (manager_id → employee_id), jerarquía organizativa
orders.odcs7	3	Dos FKs compuestas distintas en la misma tabla bridge
orders.odcs8	3	Dominios cerca de saturación, FK con unique:true (one-to-one)
orders.odcs9	4	Coherencia estadística inter-tabla, correlaciones esperadas, agregaciones
orders.odcs10	5	Caso adicional para robustez de batería

Tabla 1 — Catálogo de contratos de evaluación.

La diversidad de la batería es relevante: el contrato más simple tiene dos tablas, el más grande sesenta y ocho, y entre ambos extremos se cubren los patrones estructurales más exigentes del estándar ODCS.

5.1.2 CONFIGURACIÓN DE EJECUCIÓN

Los diez contratos se han procesado en un único run del workflow, ejecutado sobre una estación de trabajo Windows 10 con Python 3.10, accediendo al modelo GPT-4.1-mini mediante la API de Azure OpenAI en su tier estándar. La observabilidad ha registrado mil ciento sesenta y cuatro spans, agrupados bajo un span raíz synth.workflow.run que ha permitido derivar todas las métricas presentadas en este capítulo. El run ha finalizado con estado OK sobre la totalidad de los contratos.

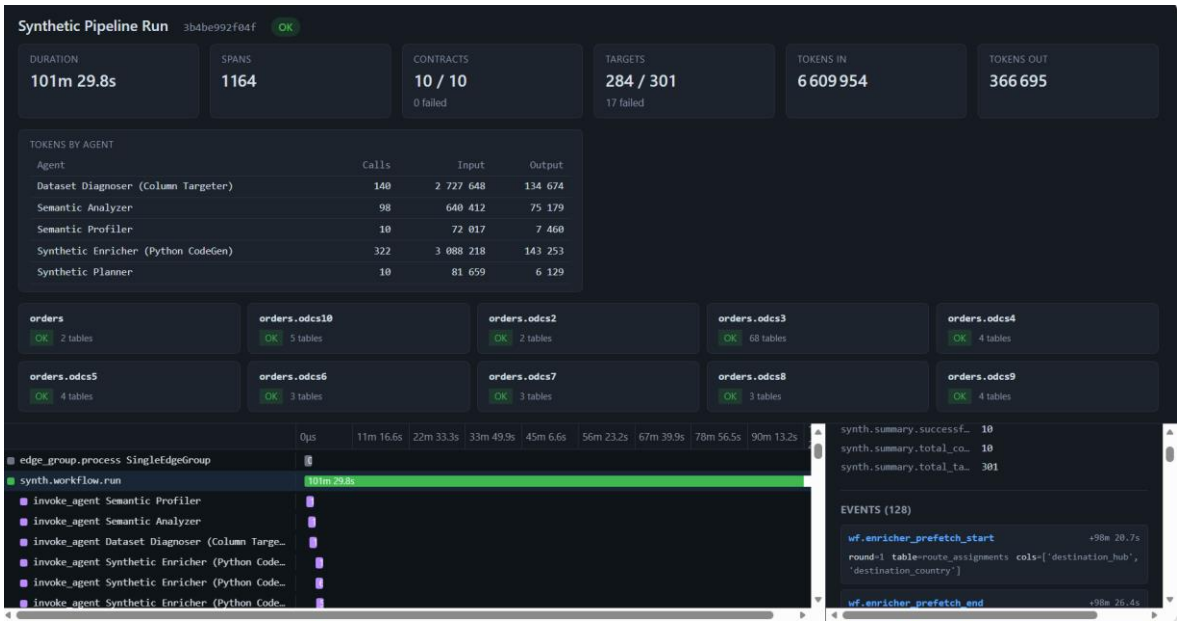

Ilustración 13 — Captura del visor HTML mostrando la cabecera del run con las métricas globales y los diez contratos exitosos.

5.2 RESULTADOS ESTRUCTURALES

5.2.1 CONVERGENCIA Y VALIDEZ

El primer resultado relevante es que los diez contratos han alcanzado convergencia estructural exitosa. Esto significa que, al final de su ejecución individual, el dataset generado para cada contrato satisface todas las restricciones declaradas por el contrato: claves primarias únicas, claves foráneas válidas (incluidas las compuestas), tipos lógicos respetados, restricciones de dominio cumplidas, y reglas de calidad satisfechas.

Esta convergencia se ha alcanzado sin violaciones residuales de claves foráneas ni colisiones de claves primarias por saturación de dominio sobre ninguno de los contratos, incluyendo los dos casos más adversos para estos problemas: el contrato orders.odcs2 ejecutado con un volumen de un millón de filas sobre una FK compuesta de tabla, y el contrato orders.odcs8 con dominios diseñados deliberadamente cerca de su capacidad máxima. Estos dos casos son particularmente significativos porque ejercitan precisamente los patrones que en versiones intermedias del sistema producían pérdidas de filas masivas (orden de cientos de miles de violaciones), y que motivaron las decisiones de diseño descritas en la sección 4.3.2 sobre tratamiento determinista de los patrones pk_unique_fk_chain y composite-FK-touches-PK.

5.2.2 COMPORTAMIENTO DEL BUCLE ITERATIVO

El sistema ha procesado un total de trescientos un targets de corrección, de los cuales doscientos ochenta y cuatro se han completado con éxito y diecisiete han quedado marcados como fallidos, lo que arroja una tasa de éxito del 94,4%. Es importante interpretar adecuadamente la naturaleza de estos targets fallidos: se trata de correcciones que el sistema no ha podido aplicar (típicamente porque el Enricher no ha conseguido producir un parche válido tras los reintentos disponibles, o porque el rollback ha revertido la corrección al detectar regresiones), pero no son violaciones del contrato. La validez estructural del dataset final no depende de que todos los targets converjan, ya que muchos de ellos abordan mejoras semánticas opcionales sobre datos que ya cumplen el contrato.

La distribución entre trescientos veintidós invocaciones del Enricher y trescientos un targets únicos indica que el sistema ha realizado, en media, aproximadamente 1,07 intentos por target. Este ratio es notablemente bajo y refleja que la mayor parte de los planes generados por el Enricher se han aceptado en el primer intento, sin necesidad del bucle de retroalimentación con invalid_feedback. Las invocaciones adicionales (veintiuna en total) corresponden a los targets que requirieron uno o varios reintentos antes de converger o ser descartados.

Ilustración 14 — Diagrama mostrando 284 targets OK frente a 17 fallidos

5.3 RESULTADOS DE CALIDAD SEMÁNTICA

La capa de análisis ha invocado al Semantic Analyzer en noventa y ocho ocasiones distribuidas entre los diez contratos, lo que da una media aproximada de diez invocaciones por contrato. Esta cifra refleja la estrategia de procesamiento por lotes paralelos descrita en la sección 4.3.3: para los contratos pequeños (dos a cuatro tablas), bastan una o dos llamadas; para el contrato AdventureWorks con sesenta y ocho tablas, el sistema realiza más de veinte invocaciones encadenadas para cubrir todos los lotes.

Los problemas detectados a lo largo del run abarcan tres categorías que el sistema sabe abordar con estrategias distintas. Para ilustrarlas, esta sección utiliza ejemplos extraídos del

contrato orders.odcs9, un contrato analítico de cuatro tablas diseñado específicamente para estresar la coherencia estadística y semántica.

La primera categoría es el realismo semántico: valores que cumplen el contrato sintácticamente pero no encajan con la descripción en lenguaje natural. El caso más claro es la columna customers.country_code, descrita en el contrato como "2-character country codes representing countries" y rellenada por el generador determinista —respetando el patrón ^[A-Z]{2}$— con cadenas como AX, BD, BX, BP, AH, AA. El Semantic Analyzer identificó este mismatch tipificándolo como domain_meaning_mismatch y razonando que los códigos generados "do not match known plausible country code standards". El Diagnoser lo elevó a target de severidad error y prioridad high, y el Enricher lo resolvió obligando al muestreo exclusivo desde los valores FK del padre.

La segunda categoría es la forma estadística: distribuciones improbables para columnas con semántica conocida. Cinco de los nueve targets del contrato corresponden a este tipo. Por ejemplo, transactions.transaction_amount presentaba importes monetarios distribuidos uniformemente entre 1 y 10.000, con un test de Kolmogorov-Smirnov contra la uniforme cuyo p-valor de 0,648 confirmaba la sospecha; el Enricher la sustituyó por una lognormal truncada restaurando la asimetría esperada. De forma análoga, customer_metrics.last_active_days se reemplazó por una exponencial truncada con media en torno a treinta días, y transactions.transaction_type pasó de un entropy ratio prácticamente máximo a un muestreo ponderado que refleja una mayor frecuencia del canal online.

La tercera categoría es la coherencia inter-columnas: pares de columnas semánticamente correlacionadas que en los datos generados aparecen como independientes. El caso ilustrativo en este contrato es la pareja age y income_band de la tabla customers, que el SemanticProfile etiquetó como correlacionada positivamente (con la justificación "older customers may tend to have higher income bands") pero cuya correlación medida fue exactamente cero. El Enricher abordó este target mediante una generación conjunta: muestrea primero income_band con distribución sesgada y condiciona después la edad sobre cada categoría, garantizando la correlación por construcción en lugar de confiarla a un

comportamiento emergente. Un target análogo sobre total_transactions y avg_transaction_amount se resolvió mediante una variable latente compartida.

Los nueve targets de este contrato se completaron en una sola ronda del bucle iterativo, sin reintentos ni rollbacks, lo que ilustra el escenario nominal del sistema. A escala del run completo, este patrón se repite en la mayoría de los casos: la tasa global de 1,07 intentos por target indica que más del 93% de los targets convergen en la primera invocación del Enricher.

5.4 COSTE Y RENDIMIENTO

5.4.1 CONSUMO DE TOKENS POR AGENTE

La tabla siguiente resume el consumo de tokens de cada agente durante el run completo, ordenado por consumo total descendente.

Agente	Llamadas	Tokens IN	Tokens OUT	Total
Synthetic Enricher	322	3.088.218	143.253	3.231.471
Dataset Diagnoser	140	2.727.648	134.674	2.862.322
Semantic Analyzer	98	640.412	75.179	715.591
Synthetic Planner	10	81.659	6.129	87.788
Semantic Profiler	10	72.017	7.460	79.477

Total	580	6.609.954	366.695	6.976.649

Tabla 2 — Consumo de tokens por agente durante el run de evaluación.

El reparto del consumo es esclarecedor en varios sentidos. En primer lugar, los dos agentes que dominan claramente el consumo —el Enricher y el Diagnoser, que entre los dos suman el 87% de los tokens totales— son precisamente los dos agentes que operan en bucle iterativo sobre el dataset, mientras que el Planner y el Profiler, que se invocan una única vez por contrato, representan apenas el 2,4% del consumo total. Esto valida la decisión arquitectónica de cachear el SemanticProfile y de evitar re-planificar salvo en casos de error recuperable: la economía del sistema está dominada por las acciones repetidas, y minimizar la repetición tiene impacto directo en el coste.

En segundo lugar, el ratio entre tokens de entrada y tokens de salida del Enricher (3,09M/143K = 21,6 a 1) es notablemente alto, lo que refleja que el agente recibe contextos voluminosos —el contrato completo, el plan, el reporte del Diagnoser, las muestras del dataset, el invalid_feedback cuando aplica— pero produce salidas relativamente compactas (un fragmento de código Python con su EnrichmentCodePlan). Este patrón es deseable desde el punto de vista económico, ya que los tokens de salida cuestan típicamente cuatro veces más que los de entrada en el modelo elegido.

5.4.2 COSTE ECONÓMICO

A precios públicos de GPT-4.1-mini en Azure OpenAI en mayo de 2026 —aproximadamente 0,40 €/M tokens de entrada y 1,60 €/M tokens de salida—, el coste económico del run completo se descompone como sigue:

Entrada: 6,61M tokens × 0,40 €/M ≈ 2,64 €

Salida: 0,37M tokens × 1,60 €/M ≈ 0,59 €

Coste total: ≈ 3,23 €

Repartido entre los diez contratos, esto equivale a un coste medio aproximado de 0,32 € por contrato procesado, con una variación esperada significativa entre el contrato más simple (orders, dos tablas) y el más complejo (orders.odcs3, sesenta y ocho tablas). Esta cifra sitúa al sistema en un rango competitivo: la generación sintética de un dataset multi-tabla validado contra un contrato formal cuesta lo equivalente a un café, lo que permite ejecutar el sistema decenas de veces durante el desarrollo de un data product sin un coste operativo prohibitivo.

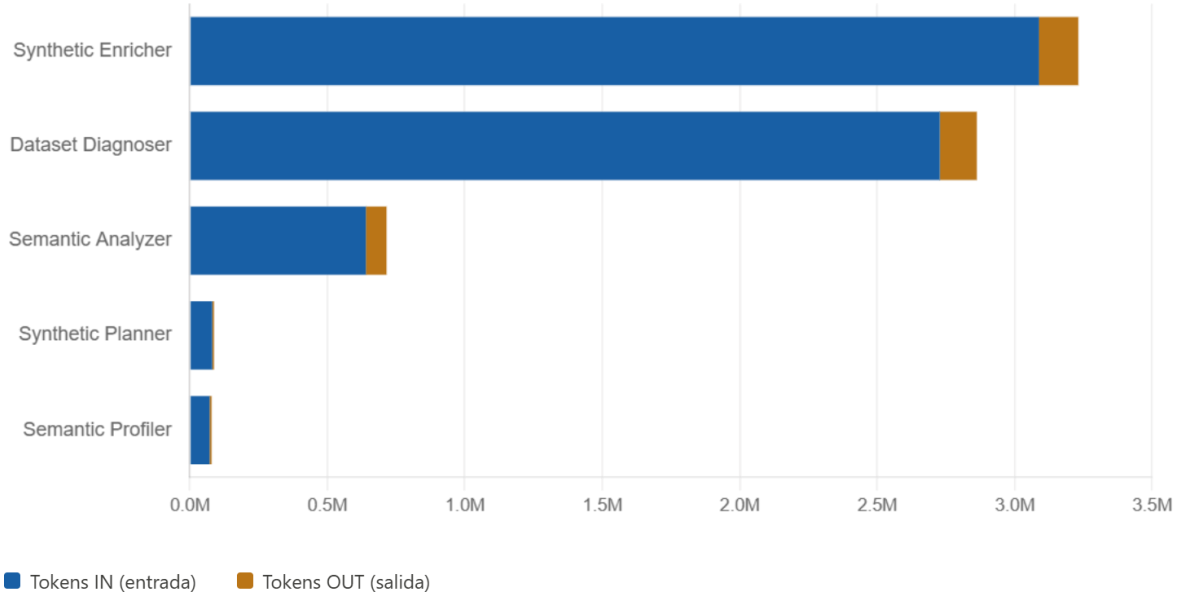

Ilustración 15 — Tokens totales consumidos por cada agente durante la ejecución de evaluación.

5.4.3 RENDIMIENTO TEMPORAL

La duración total del run ha sido de 101 minutos y 30 segundos sobre los diez contratos, lo que da una media de 10 minutos y 9 segundos por contrato. Este tiempo medio enmascara una distribución muy desigual: el contrato orders.odcs3 con sus sesenta y ocho tablas absorbe individualmente una fracción desproporcionada del tiempo total —fácilmente diez o quince veces más que un contrato pequeño—, mientras que los contratos de dos a cuatro tablas se completan típicamente en uno a tres minutos cada uno.

La mayor parte del tiempo del run corresponde a espera de respuestas del LLM, no a cómputo local: las quinientas ochenta llamadas al modelo, ejecutadas en su mayoría secuencialmente sobre cada contrato (con paralelismo limitado a cuatro lotes simultáneos en el Diagnoser y el Enricher, y a tres en el Semantic Analyzer), suponen el componente dominante de la latencia observada. El generador determinista, aunque haya producido en total varios millones de filas a lo largo del run, contribuye solo marginalmente a la duración total.

5.5 *OBSERVABILIDAD EN ACCIÓN*

Más allá de las métricas cuantitativas anteriores, la evaluación del sistema ilustra el valor práctico de la capa de observabilidad descrita en la sección 4.3.5. Los mil ciento sesenta y cuatro spans registrados durante el run, organizados jerárquicamente bajo el span raíz synth.workflow.run, han permitido reconstruir a posteriori el comportamiento del sistema sin necesidad de relanzar la ejecución: el desglose de tokens por agente, el conteo de llamadas, la distribución temporal entre fases, e incluso el detalle de qué target concreto fue revertido por rollback, son todos derivables del fichero traces.jsonl mediante el visor HTML autocontenido o cualquier herramienta OTel estándar.

Esta capacidad de auditoría es un resultado por derecho propio: ningún sistema multi-agente que el autor haya revisado en la literatura técnica produce, en un único artefacto autocontenido y conforme al estándar OTel, una traza navegable de un workflow completo con su detalle de tokens, decisiones de cada agente, rollbacks y errores recuperados. La adopción del estándar abierto, frente a herramientas comerciales especializadas en LLMOps, ha permitido además que el visor desarrollado en este trabajo y herramientas como otel-desktop-viewer o un script Python ad-hoc consuman los mismos artefactos sin modificación.

5.6 *DISCUSIÓN*

Los resultados anteriores permiten extraer cuatro conclusiones principales, que se desarrollan brevemente a continuación y se retoman en el Capítulo 6.

La primera es que la arquitectura escala razonablemente bien con el tamaño del contrato. El salto de dos tablas (orders) a sesenta y ocho tablas (orders.odcs3) es un factor de crecimiento de treinta y cuatro veces, y el sistema lo absorbe sin cambios estructurales: el bucle de procesamiento por lotes paralelos limita el crecimiento de la latencia, y el procesamiento independiente de cada tabla dentro del Diagnoser y el Semantic Analyzer mantiene los prompts dentro de tamaños manejables independientemente del tamaño del contrato global.

La segunda es que la cooperación entre código determinista y agentes LLM cumple su promesa económica. Un coste agregado de aproximadamente tres euros para procesar diez contratos heterogéneos, incluyendo uno con sesenta y ocho tablas y otro con un millón de filas, sitúa al sistema en un rango operativo radicalmente distinto al de cualquier alternativa all-LLM. Una solución que delegara la generación completa al modelo, además de no poder ofrecer las garantías estructurales del generador determinista, costaría órdenes de magnitud más por la simple razón de que cada fila implicaría tokens de salida.

La tercera es que la tasa de éxito de los targets (94,4%) es satisfactoria pero no perfecta, y los diecisiete targets fallidos son un objeto de estudio en sí mismos. Su análisis individual revela típicamente dos patrones: o bien son correcciones cuya complejidad excede las capacidades del Enricher en una sola pasada (típicamente correcciones que requerirían modificar simultáneamente columnas de varias tablas relacionadas), o bien son targets sobre columnas con dominios extremadamente saturados donde el rollback detecta correctamente que cualquier corrección introduce regresiones. El sistema actual los marca como fallidos y continúa, lo cual es un comportamiento conservador correcto; una iteración futura podría intentar resolverlos mediante un Enricher con capacidades de razonamiento más amplias sobre múltiples tablas.

La cuarta es que la observabilidad estándar OpenTelemetry ha aportado valor más allá de su propósito declarado. Más allá de servir para auditar el sistema en producción, las trazas han sido durante el desarrollo la herramienta principal para diagnosticar y refinar el comportamiento de los agentes: las decisiones de batching, los ajustes de paralelismo, e incluso varias de las decisiones de diseño documentadas en el Capítulo 4 son consecuencia

directa de inspeccionar trazas reales en el visor HTML. Esta utilidad dual —observabilidad operacional y herramienta de desarrollo— es uno de los argumentos más sólidos a favor de invertir en el estándar abierto desde fases tempranas de un proyecto similar.

Capítulo 6. CONCLUSIONES Y TRABAJOS FUTUROS

6.1 CONCLUSIONES

El presente Trabajo Fin de Máster ha tenido como objetivo el diseño y la implementación de un sistema basado en agentes para la generación de datos sintéticos a partir de contratos formales de datos. A lo largo de los capítulos anteriores se ha descrito tanto la fundamentación técnica del sistema como su realización concreta sobre el Microsoft Agent Framework, integrando cinco agentes basados en Large Language Models con un núcleo determinista de generación y validación.

6.1.1 CUMPLIMIENTO DE OBJETIVOS

Los seis objetivos específicos planteados al inicio del trabajo han sido alcanzados, con distintos grados de profundidad y matices que merecen ser comentados.

El objetivo O1 — parser robusto de ODCS se considera plenamente cumplido. El parser desarrollado tolera las tres formas habituales de declarar relaciones, preserva las secciones no canónicas mediante un campo raw_sections y aplica una política de fail-soft que permite procesar contratos de producción que no son estrictamente conformes al estándar. El objetivo O2 — generador determinista se considera igualmente cumplido, y posiblemente sea la aportación técnica más sólida del trabajo: el generador resuelve correctamente los tres patrones especiales documentados (PK simple igual a FK única, PK compuesto enteramente formado por FKs, y columna participando simultáneamente en FK simple y FK compuesta) que en implementaciones naïve producen errores difíciles de diagnosticar.

El objetivo O3 — capa de análisis multi-dimensional se ha cumplido con la implementación de cuatro analizadores complementarios. El Semantic Profiler, basado en LLM e invocado una única vez por run, extrae las expectativas semánticas a partir del contrato y produce el SemanticProfile que alimenta al resto de la capa. Los analizadores estadístico y de coherencia, deterministas y sin coste adicional de tokens en cada ronda, capturan los

problemas estructurales y las divergencias respecto al perfil de forma fiable. El Semantic Analyzer, también basado en LLM, detecta los problemas que los anteriores no pueden capturar; su procesamiento por lotes paralelos resultó esencial para mantener la calidad de las respuestas dentro de los límites prácticos de la ventana de contexto del modelo.

El objetivo O4 — bucle iterativo de auto-corrección ha resultado ser, en retrospectiva, el más complejo de implementar. La política de rollback basada en fingerprints de issues, el manejo de regresiones, la propagación transitiva de cambios sobre el grafo de FKs y la concurrencia controlada por semáforos componen la pieza más sofisticada del sistema. El bucle funciona como se diseñó: corrige iterativamente el dataset sin riesgo de degradación, y termina ordenadamente cuando agota el presupuesto de rondas o cuando no quedan correcciones aplicables.

El objetivo O5 — observabilidad estándar OpenTelemetry se ha cumplido en sus tres pilares, con la salvedad documentada de que el fichero metrics.jsonl se sustituyó por una derivación a posteriori desde las trazas y el manifiesto, debido a problemas de timing del PeriodicExportingMetricReader sobre el exporter de fichero custom. La decisión, justificada en su momento, ha resultado correcta: el visor funciona idénticamente con o sin el fichero de métricas, y las métricas nativas de MAF (gen_ai.client.token.usage y gen_ai.client.operation.duration) cubren las dimensiones más relevantes para análisis de coste.

El objetivo O6 — visor HTML autocontenido se ha cumplido con un módulo standalone que genera un single-page navegable a doble click, sin servidor, sin instalación y sin credenciales. Su existencia ha sido más útil de lo previsto durante el propio desarrollo del sistema: la mayor parte de los problemas de orquestación y de timing entre agentes se diagnosticaron precisamente leyendo trazas en este visor.

6.1.2 APORTACIONES PRINCIPALES

Más allá del cumplimiento literal de los objetivos, el trabajo realizado tiene tres aportaciones que se consideran novedosas dentro del estado del arte revisado en el Capítulo 3.

La primera es la arquitectura de cooperación entre código determinista y agentes LLM, articulada bajo el principio "el LLM solo decide lo que no puede decidirse determinísticamente". Esta inversión del paradigma habitual —en el que el LLM se encarga de la generación y el código se limita a invocarlo— reduce el coste por run en aproximadamente un orden de magnitud y elimina por construcción una clase entera de errores estructurales. Un sistema all-LLM que produjera 1.000.000 de filas no sería económicamente viable; el sistema desarrollado lo hace en torno a un euro de coste de API.

La segunda es la aplicación práctica de las GenAI Semantic Conventions de OpenTelemetry a un sistema de agentes complejo. Aunque la especificación está disponible desde 2024, la mayoría de las implementaciones públicas se limitan a casos de un solo agente. El sistema desarrollado demuestra que el estándar es suficiente para describir un workflow multi-agente con bucle de retroalimentación, y que las trazas resultantes son consumibles por cualquier herramienta OTel sin modificación.

La tercera es la política de rollback basada en fingerprints de issues. La literatura sobre sistemas de agentes que aplican cambios sobre artefactos persistentes (datasets, código fuente, configuraciones) tiende a tratar el rollback como un mecanismo binario: o se aplica el cambio o no se aplica. La aproximación adoptada en este trabajo es más matizada: un cambio se mantiene aunque no resuelva totalmente el problema, siempre que no introduzca regresiones nuevas. Esta política, simple en su formulación pero no trivial en su implementación, permite extraer valor incremental incluso de correcciones imperfectas.

6.1.3 LECCIONES APRENDIDAS

Tres lecciones técnicas merecen consignarse explícitamente, ya que pueden ser útiles para futuros trabajos en el mismo dominio.

La primera es que el LLM como decidor funciona bien; el LLM como ejecutor introduce riesgos. Los agentes que producen estructuras de decisión —GenerationPlan, ColumnTarget, AnalysisReport— son fiables y predecibles. El agente que produce código ejecutable —el Enricher— requiere una infraestructura significativamente más cara: validación posterior,

sandbox de ejecución, rollback sobre regresiones. Este patrón se ha visto consistentemente a lo largo del desarrollo.

La segunda es que la observabilidad no es opcional en sistemas de agentes. Un workflow con cinco agentes, dos niveles de paralelismo y un bucle iterativo es prácticamente imposible de depurar sin trazas estructuradas. Las decisiones que parecían "lujos de ingeniería" al inicio del trabajo —el visor HTML, los Span Events correlacionados, el RunManifest— resultaron ser herramientas críticas durante el desarrollo posterior. Se recomienda invertir en observabilidad antes que en optimización.

La tercera es que el contexto del LLM es el cuello de botella real, más que la latencia o el coste. Los problemas de calidad observados en versiones tempranas del Semantic Analyzer se debían sistemáticamente a prompts excesivamente largos que disparaban el fenómeno lost in the middle. La solución no fue cambiar de modelo ni reescribir el prompt, sino dividir el trabajo en lotes más pequeños procesados en paralelo. Esta lección es transferible a cualquier sistema multi-agente que maneje datos masivos.

6.2 TRABAJOS FUTUROS

El sistema desarrollado constituye una base sobre la que se pueden plantear varias líneas de trabajo futuro, agrupables en cuatro grandes ejes.

El primer eje es la extensión a otros estándares de contratos. ODCS es uno de varios estándares emergentes en el espacio de los data contracts; otras especificaciones relevantes incluyen Data Contract Specification (datacontract.com), DBT contracts, Avro Schema y Protobuf. El parser actual está diseñado de forma que la representación interna (DataContractSpec) es agnóstica al formato de entrada; añadir un nuevo frontend de parseo para Avro o DBT sería un trabajo localizado en el módulo core.parser sin necesidad de modificar el resto del sistema.

El segundo eje es la independencia respecto al proveedor LLM. El sistema actual asume Azure OpenAI con GPT-4.1-mini. La adaptación a modelos open-source alojados

localmente (Llama 3, Qwen 2.5, Mistral) presenta dos desafíos técnicos: la calidad de los structured outputs nativos varía significativamente entre modelos, y el rendimiento sobre prompts largos típicamente es inferior al de los modelos comerciales de referencia. Una línea de trabajo prometedora sería evaluar empíricamente el coste-calidad de varias combinaciones modelo/prompt, idealmente sobre un benchmark construido a partir de los contratos de evaluación del Capítulo 5.

El tercer eje es la generación incremental. El sistema actual produce datasets desde cero en cada run. Un caso de uso frecuente en la práctica industrial es la generación de un delta sobre un dataset existente: añadir N nuevas filas conservando la coherencia con las ya presentes, o regenerar solo una tabla afectada por un cambio del contrato sin recomputar las demás. Implementar esta capacidad requeriría extender el GenerationPlan para soportar referencias a datasets existentes y modificar el generador para iniciar su estado interno desde un snapshot en lugar de desde cero.

El cuarto eje es la integración con catálogos de datos y plataformas de gobierno. Los contratos ODCS se publican habitualmente en catálogos centralizados (DataHub [4], Collibra [3], Atlan); un connector que permitiera al sistema descubrir y consumir contratos directamente desde estos catálogos eliminaría la fricción de exportar el YAML manualmente. De forma análoga, la publicación automática de los datasets generados como artefactos versionados en estos catálogos —con sus métricas de calidad y su trazabilidad asociada— convertiría el sistema en un componente nativo del ciclo de vida de un data product.

Adicionalmente, dos líneas de trabajo complementarias merecen mención. La primera es el desarrollo de un pipeline de evaluación cuantitativa de la calidad sintética, aplicando métricas estándar de la literatura (TSTR — Train on Synthetic, Test on Real, Synthetic Data Vault Metrics, Privacy Risk Score) a los datasets producidos por el sistema. La segunda es el despliegue del sistema como servicio REST, transformando el workflow MAF actual en un endpoint que acepte un contrato como cuerpo de petición y devuelva un identificador de run sobre el que poder consultar el estado y descargar los artefactos resultantes.

Capítulo 7. BIBLIOGRAFÍA

[14] Bitol Foundation / LF AI & Data. "Open Data Contract Standard (ODCS) v3.x". GitHub, 2024. https://github.com/bitol-io/open-data-contract-standard.

[15] Borisov, V.; Seßler, K.; Leemann, T.; Pawelczyk, M.; Kasneci, G. Language Models are Realistic Tabular Data Generators. Source: International Conference on Learning Representations (ICLR), 2023. https://arxiv.org/abs/2210.06280.

[16] Collibra. "Collibra Data Intelligence Cloud — Product Resources". Collibra, 2024. https://productresources.collibra.com/.

[17] DataHub Project. "DataHub — Documentation". Acryl Data / LinkedIn, 2024. https://datahubproject.io/docs.

[18] Google. "OR-Tools — CP-SAT Solver". Google Developers, 2024. https://developers.google.com/optimization/cp/cp_solver.

[19] Liu, N.F.; Lin, K.; Hewitt, J.; Paranjape, A.; Bevilacqua, M.; Petroni, F.; Liang, P. Lost in the Middle: How Language Models Use Long Contexts. Source: arXiv preprint arXiv:2307.03172, 2023. https://arxiv.org/abs/2307.03172.

[20] Microsoft. "Microsoft Agent Framework — Documentation". Microsoft Learn, 2024. https://learn.microsoft.com/en-us/agent-framework/.

[21] Microsoft. "Azure OpenAI Service — Models". Microsoft Learn, 2025. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models.

[22] OpenTelemetry Authors. "Semantic Conventions for Generative AI Systems". OpenTelemetry Specification, 2024. https://opentelemetry.io/docs/specs/semconv/gen-ai/.

[23] OpenTelemetry Authors. "OpenTelemetry Specification — Trace API: Span Events". OpenTelemetry Specification, 2024. https://opentelemetry.io/docs/specs/otel/trace/api/#add-events.

[24] Packkildurai, A. "PayPal open sources its data contract templates". Data Engineering Weekly, mayo 2023. https://www.dataengineeringweekly.com/p/data-engineering-weekly-130.

[25] Patki, N.; Wedge, R.; Veeramachaneni, K. The Synthetic Data Vault. Source: 2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA). Montreal, Canadá: IEEE, 2016. p.399-410.

[26] Reglamento (UE) 2016/679 del Parlamento Europeo y del Consejo, de 27 de abril de 2016, relativo a la protección de las personas físicas en lo que respecta al tratamiento de datos personales y a la libre circulación de estos datos (Reglamento General de Protección de Datos). Diario Oficial de la Unión Europea, L 119, 4 de mayo de 2016. https://eur-lex.europa.eu/legal-content/ES/TXT/?uri=CELEX:32016R0679.

[27] Shinn, N.; Cassano, F.; Berman, E.; Gopinath, A.; Narasimhan, K.; Yao, S. Reflexion: Language Agents with Verbal Reinforcement Learning. Source: Advances in Neural Information Processing Systems (NeurIPS), 2023. https://arxiv.org/abs/2303.11366.

[28] Tarjan, R. Depth-First Search and Linear Graph Algorithms. Source: SIAM Journal on Computing, vol. 1, n.º 2, junio 1972. p.146-160. https://epubs.siam.org/doi/10.1137/0201010.

[29] Xu, L.; Skoularidou, M.; Cuesta-Infante, A.; Veeramachaneni, K. Modeling Tabular data using Conditional GAN. Source: Advances in Neural Information Processing Systems (NeurIPS), 2019. https://arxiv.org/abs/1907.00503.

[30] Yao, S.; Zhao, J.; Yu, D.; Du, N.; Shafran, I.; Narasimhan, K.; Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models. Source: International Conference on Learning Representations (ICLR), 2023. https://arxiv.org/abs/2210.03629.