Design Strategies for Multi-Agent Systems: Lessons from Four Implementations

Covarrubias Lucía (0009-0001-5782-4145), Rivero Tomás (0009-0007-7092-0629),
Bosch Ignacio (0009-0006-0690-7043)
I+D+i - C&S
lcovarrubias@cys.com.ar, trivero@cys.com.ar, ibosch@cys.com.ar
Junio, 2026

Abstract. Orchestrating multi-agent systems involves a series of concrete design decisions. Components such as Retrieval-Augmented Generation (RAG), external tools, heterogeneous memory stores, security policies, and protocols like the Model Context Protocol (MCP) must be integrated coherently. Recurring questions arise in practice: which vector database to use, when to delegate tasks between agents, and when to expose one agent as a tool for another. This paper presents a retrospective analysis of four systems deployed in real environments: NeuroHire, Doki, MateApp, and Ciruelito, exploring different orchestration approaches using the OpenAI Agents SDK and the LangChain framework. From this analysis, ten design strategies were identified, among which the following stand out: hierarchical agent delegation, RAG with explicit distance-to-similarity conversion and threshold filtering in FAISS, Quota-Driven Degradation with automatic fallback to a cheaper model upon exceeding usage limits, and MCP Tool Filter to reduce the attack surface of exposed tool sets. These strategies address problems observed during development: irrelevant context in RAG pipelines, cost management in multi-tenant systems, security in tool integration and data isolation, and improved maintainability through encapsulation and technical guardrails. The strategies are transferable to any domain combining LLMs with structured knowledge sources, enterprise tools, and business policies.

Keywords: multi-agent systems, Agents SDK, RAG, MCP, prompt engineering.

Estrategias de Diseño para Sistemas Multiagente: Lecciones de Cuatro Implementaciones

Resumen. La orquestación de sistemas multiagente exige decisiones de diseño. Componentes como la generación aumentada por recuperación (RAG), herramientas externas, memorias heterogéneas, políticas de seguridad y protocolos como Model Context Protocol (MCP) deben integrarse de forma coherente. Esto plantea interrogantes concretas: qué base de datos vectorial utilizar, cuándo delegar tareas entre agentes y cuándo exponer un agente como herramienta de otro. Este trabajo presenta un análisis retrospectivo de cuatro sistemas implementados en distintos entornos: NeuroHire, Doki, MateApp y Ciruelito. En estos sistemas se exploraron distintas formas de orquestación con el OpenAI Agents SDK y el framework LangChain. A partir del análisis se observaron diez estrategias de diseño, entre las que se destacan: la delegación jerárquica entre agentes, RAG con umbralización y conversión explícita de distancia a similitud en FAISS, Quota-Driven Degradation con degradación a un modelo más económico tras superar el límite de uso, y MCP Tool Filter para reducir la superficie de ataque. Estas estrategias abordan

problemas observados durante el desarrollo: contexto irrelevante en RAG, gestión de costos en entornos multi-tenant, seguridad en la integración de herramientas, aislamiento de datos y mejora de la mantenibilidad mediante encapsulación y guardrails técnicos.

Palabras clave: sistemas multiagente, Agents SDK, RAG, MCP, ingeniería de prompts.

1. Introducción

La aparición de los grandes modelos de lenguaje (LLM por sus siglas en inglés) hizo posible una nueva clase de sistemas de software en los que el modelo opera como núcleo de un ciclo de razonamiento y acción, capaz de invocar herramientas, recuperar conocimiento externo y coordinarse con otros agentes (Schick et al., 2024; Yao et al., 2023). El concepto de agente inteligente tiene una larga trayectoria en la literatura (Russell and Norvig, 2004), pero la irrupción de los LLMs dan un nuevo significado al término agente de manera sustantiva. Esto debido a que los LLMS incorporan conocimiento enciclopédico extenso y adaptan su salida en formato y tono según el contexto de la interacción. Además, la aparición de frameworks como el de OpenAI Agents SDK (OpenAI, 2025) reducen la dificultad de implementación de estas tecnologías, al proveer abstracciones para la creación de agentes, la asignación de roles, el uso de herramientas, la trazabilidad de ejecuciones y la integración de protocolos de contexto como el MCP (Anthropic et al., 2024).

Sin embargo, la distancia entre la teoría de sistemas multiagente y la práctica de ingeniería persiste. Las publicaciones existentes abordan estos sistemas desde perspectivas taxonómicas o de benchmark (Guo et al., 2024; Li et al., 2024), pero las decisiones de diseño concretas que emergen en implementaciones reales rara vez se documentan con el nivel de detalle necesario para ser reproducibles. Decisiones como la elección entre la transferencia de control entre agentes (handoffs) frente a la exposición de un agente como herramienta de otro mediante `agent.as_tool`, la gestión de estado en entornos con múltiples usuarios simultáneos, el control de costos operativos mediante cuotas, o el aislamiento de índices de búsqueda semántica por usuario, tienen un impacto directo sobre la seguridad, el costo y la experiencia del usuario del sistema. Sin embargo, no cuentan con guías de referencia consolidadas. Esta brecha es el motivo principal del presente trabajo.

Se presenta en este trabajo un análisis retrospectivo del desarrollo de cuatro sistemas de sistemas multiagente basados en LLM: NeuroHire para el reclutamiento inteligente con análisis de video y audio, Doki para asistencia de documentación específica con sistema multi-tenant en tiempo real, MateApp para asistencia académica de manera administrativa y pedagógica, y por ultimo, Ciruelito para investigación con fuentes híbridas e integración con Azure DevOps. Todos estos proyectos son experimentales en entornos de desarrollo, en el marco de actividades de I+D y sirvieron como base fundamental para el diseño y observación de estrategias que están documentadas en este trabajo. El análisis de los cuatro sistemas, permitió identificar un conjunto de diez estrategias de diseño que se presentan como observaciones derivadas emergentes de los proyectos mencionados.

La pregunta que orienta este trabajo es sobre ¿qué decisiones de diseño emergen de manera recurrente al implementar sistemas multiagente basados en LLMs en estos

entornos, y qué lecciones pueden extraerse de su comparación? La respuesta a esta incógnita se resolvió de forma empírica mediante el relevamiento de los cuatro sistemas, extrayendo patrones de implementación.

El trabajo se organiza de la siguiente manera: en la sección 2 se introduce el marco teórico sobre los temas que cubre este trabajo, en la sección 3 se revisa el estado del arte, la sección 4 describe la metodología junto con las estrategias de diseño para luego dar paso a la sección 5 donde se detallan los proyectos, en la sección 6 se trasmiten los resultados obtenidos y en la sección 7 la discusión de los mismos, y en la sección 8 se cierra el trabajo.

2. Marco teórico

2.1. Generación Aumentada por Recuperación

La generación aumentada por recuperación (RAG, por sus siglas en inglés) es una estrategia que permite a un LLM acceder a información externa específica para la cual no fue entrenado. El objetivo es mejorar la calidad de las respuestas generadas mediante el aporte de contexto relevante recuperado de documentos, mitigando así el problema de las alucinaciones sin necesidad de realizar el *fine-tuning* del modelo (Lewis et al., 2020).

Una implementación práctica de RAG implica tomar varias decisiones de diseño, entre las que se destacan: la estrategia de particionado del texto (chunking) y el solapamiento entre particiones, la elección del modelo de vectorización (embeddings), la métrica de similitud a utilizar en la búsqueda (similitud coseno, Maximal Marginal Relevance, entre otras), el valor del umbral de recuperación y la gestión de la base vectorial (vectorstore). Cada una de estas decisiones afecta la relevancia del contexto recuperado y, por lo tanto, la calidad de la respuesta generada. Una característica que puede pasar desapercibida es que algunas implementaciones para bases vectoriales como FAISS (Johnson et al., 2019), devuelven distancias y no similitudes, lo que requiere una conversión explícita antes de aplicar umbrales de filtrado. Esto es porque al aplicar un umbral directamente sobre el score devuelto sin una conversión explícita se produce un filtrado inverso al buscado y se retienen los documentos menos relevantes. En la figura 1 se ilustra el flujo general de un sistema RAG.

2.2. Agents SDK

El OpenAI Agents SDK (OpenAI, 2025) es un framework que provee las abstracciones necesarias para implementar, orquestar y monitorizar sistemas agénticos. Entre los componentes relevantes para este trabajo se encuentran: Agent es la instancia de un agente a la cual es necesario definir nombre, instrucciones, modelo, herramientas y servidores MCP, Runner es el encargado de ejecutar la co-rutina del agente y recibir la petición del usuario, Trace es la forma de para registrar cada ejecución y visualizarla en la plataforma de OpenAI, SQLiteSession para persistir el historial de conversación entre peticiones y ModelSettings para configurar parámetros del modelo como la política de selección de herramientas (tool_choice).

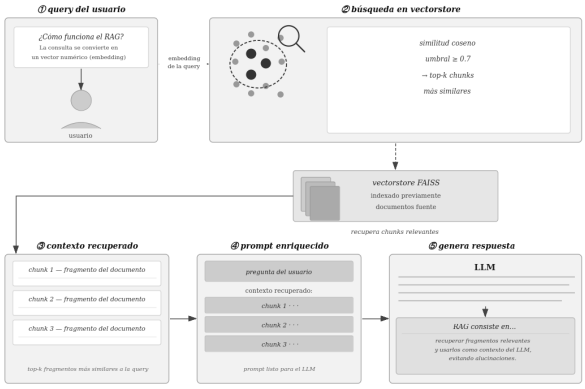

Figura 1: Flujo de un sistema RAG.

2.3. Model Context Protocol

El Model Context Protocol (MCP) (Anthropic et al., 2024) es un protocolo que estandariza el acceso a contexto externo para modelos de lenguaje. Su propósito es permitir que herramientas propias o de terceros se conecten a distintos sistemas agénticos de manera uniforme, evitando así integraciones específicas por proyecto. En la práctica, los servidores MCP pueden levantarse como procesos externos mediante MCPServerStdio y gestionarse durante el ciclo de vida de una aplicación a través de AsyncExitStack.

2.4. Tools y Handoffs

Las herramientas son el componente que distingue a un agente de un LLM simple. En sistemas con múltiples agentes, la asignación de herramientas a cada agente es una decisión de diseño relevante tanto para la seguridad como para la especialización funcional. El framework OpenAI Agents SDK ofrece tres mecanismos principales (Tabla 1): @function_tool decorador que transforma una función de Python en una herramienta útil para el agente, agent.as_tool que convierte un agente completo en una herramienta invocable por otro agente manteniendo instrucciones y modelo propios pero sin acceso al contexto del agente padre, y handoffs como mecanismo por el cual un agente transfiere el control completo de la conversación a otro agente especializado que puede interactuar directamente con el usuario.

2.5. Prompts

En sistemas agénticos, el prompt de sistema es el mecanismo principal para moldear el comportamiento de un agente sin necesidad de modificar o re-entrenar el modelo subyacente (White et al., 2024). La elaboración de prompts efectivos es un proceso iterativo y la separación de las instrucciones en archivos dedicados, independientes del código de orquestación, facilita este proceso y mejora la legibilidad del sistema. Existen muchas técnicas de ingeniería de prompt para implementar en la creación del agente, sin

Dimensión	`@function_tool`	`agent.as_tool`	`handoffs`
Razonamiento propio	No	Sí	Sí
Acceso sesión padre	No	No	Sí
Interactúa con usuario	No	No	Sí
Observabilidad	Alta (entrada/salida)	Alta (entrada/salida)	Baja
Caso típico	Lógica determinista	Subtarea encapsulada	Cambio de contexto

Tabla 1: Comparación de los mecanismos de delegación del Agents SDK.

embargo la anatomía clásica de un prompt siempre sigue la forma de: roles, contexto, directivas, formato y criterios o restricciones.

2.6. Sistemas multiagente

Un sistema multiagente puede definirse como un conjunto de entidades autónomas que perciben el entorno y actúan para alcanzar objetivos mediante algún protocolo de comunicación (Wooldridge, 2009). En la práctica, la autonomía en estos sistemas es relativa: los agentes siguen políticas establecidas mediante instrucciones, esquemas de herramientas y orquestadores. En el esquema de orquestación centralizada con delegación jerárquica (Xi et al., 2025), un agente raíz concentra la interacción con el usuario y delega tareas específicas a subagentes especializados, manteniendo el control sobre la respuesta final.

3. Estado del arte

El diseño de sistemas multiagente basados en LLM avanzó de forma notable desde el año 2023. Los trabajos en esta área pueden organizarse en torno a tres ejes: frameworks de orquestación, estrategias de recuperación de información y evaluación de calidad.

En cuanto a frameworks de orquestación, Wu et al. (2023) modelan las aplicaciones como conversaciones entre agentes configurables con participación humana en el proceso. Hong et al. (2024) proponen codificar procedimientos operativos estándar en prompts para coordinar agentes con roles especializados en pipelines secuenciales. Aquí se utiliza el framework de LangGraph que modela flujos agénticos como grafos dirigidos con estado compartido. En los estudios de investigación de Li et al. (2024) y Guo et al. (2024) se establecen taxonomías de estructuras de comunicación e identifican la orquestación como un área crítica en la creación de sistemas multiagentes.

En cuanto a estrategias de recuperación, Lewis et al. (2020) introducen RAG demostrando que recuperar pasajes relevantes en tiempo de inferencia mejora tareas intensivas en conocimiento sin necesidad de re-entrenar el modelo. El trabajo original opera sobre un índice global compartido y no aborda estrategias de filtrado por confianza de recuperación. En los sistemas presentados en este trabajo se utilizó RAG con umbral de similitud explícito sobre FAISS (Johnson et al., 2019), y en uno de ellos se incorporó

descripción de imágenes extraídas de documentos mediante LLM para indexar contenido visual junto al texto.

En cuanto a la evaluación automatizada de respuestas, Zheng et al. (2023) sistematizaron empíricamente el uso de modelos de lenguaje como jueces de otros modelos, demostrando que jueces fuertes como gpt-4 alcanzan un acuerdo superior al 80 % con preferencias humanas. Por otra parte, relacionado a sistemas multiagente aplicados a dominios específicos, Xie et al. (2025) proponen MARSHA, una arquitectura multiagente basada en RAG para adaptación frente a riesgos naturales, instanciada en WildfireGPT para incendios forestales. Por su parte, Althaf et al. (2025) presentan un marco multiagente RAG para *entity resolution* en registros de hogares, implementado sobre LangGraph con cuatro agentes especializados, que reporta 94.3 % de exactitud.

Los trabajos relevados documentan avances en frameworks, arquitecturas y aplicaciones de dominio, pero las decisiones de diseño que emergen en implementaciones concretas con restricciones operativas reales rara vez se describen con el nivel de detalle necesario para ser reproducibles. Este trabajo describe cuatro sistemas desarrollados en ese contexto y las decisiones de diseño observadas a partir de su análisis.

4. Metodología

El enfoque adoptado en este trabajo es de análisis retrospectivo sobre cuatro proyectos desarrollados entre comienzos del año 2024 y fines del 2025. El análisis es cualitativo y no comparativo en sentido estricto: no se contrastan los sistemas entre sí con métricas, sino que se identifican patrones de decisión recurrentes a lo largo de las implementaciones. Este enfoque se corresponde con el estudio de casos múltiplesYin (2018) aplicado al diseño de software.

En cada proyecto se define primero el problema que se enfrenta. Una vez establecido, se da lugar a la creación y asignación de roles de los agentes y herramientas que los integran, y finalmente las políticas de seguridad, costo y observabilidad. Estos sistemas evolucionaron a través de iteraciones cortas con ajustes de prompts, herramientas y arquitectura, basados en el comportamiento observado en cada ciclo. Este proceso iterativo es el que permitió observar las estrategias de diseño que se describen a continuación.

4.1. Estrategias de diseño

A continuación, se describen diez estrategias observadas en los sistemas analizados. No se presentan como un catálogo normativo, sino como decisiones de diseño concretas que emergieron de la práctica, con su justificación técnica. La tabla 2 muestra su presencia en cada sistema.

E1. Agent as Tool. Exponer subagentes especializados al orquestador mediante agent.as_tool reduce la superficie de decisión del orquestador a un conjunto acotado de herramientas de alto nivel, y aísla las instrucciones y el modelo de cada dominio dentro del subagente. La calidad de la descripción de cada herramienta (tool_description) es determinante: el orquestador la utiliza para decidir cuándo y cómo delegar. La limitación principal es que el subagente no puede interactuar directamente con el usuario ni acceder al contexto del agente padre.

E2. SandboxPath Injection. Fijar la ruta del sandbox del usuario mediante un ContextVar de Python antes de ejecutar el agente actúa como guardrail imperativo de parámetros. Cuando el LLM infiere incorrectamente el valor de un parámetro de ruta, el ContextVar lo sobrescribe sin requerir cambios en el modelo ni en el prompt. Esta estrategia surgió de un comportamiento recurrente observado en desarrollo y resuelve una clase de error que las instrucciones adicionales en el prompt no lograban eliminar de forma confiable.

E3. Threshold RAG. Rechazar el contexto recuperado cuya similitud semántica no supera un umbral mínimo evita que el LLM genere respuestas ancladas en documentos de baja relevancia. Un aspecto que suele omitirse es que FAISS devuelve distancias y no similitudes, por lo que aplicar un umbral directamente sobre la distancia sin conversión produce comportamientos incorrectos. El valor del umbral es empírico y dependiente del dominio, lo que indica que su calibración requiere casos representativos de cada aplicación.

Listing 1: Pseudocódigo abstracto: Threshold RAG (E3).

```
Entrada: consulta q, coleccion D, umbral theta, cantidad k
Salida:  fragmentos relevantes F

V <- vectorstore(D)
candidatos <- buscar_similitud(V, q, k)

F <- []
para cada (fragmento, distancia) en candidatos:
    similitud <- convertir(distancia)   // s = 1 - d si metrica L2
    si similitud >= theta:
        agregar fragmento a F

si F esta vacio:
    retornar sin_contexto
retornar F
```

E4. Role-Based Tooling. Definir la lista de herramientas de cada agente en su constructor, de forma cerrada, impide que el LLM invoque herramientas fuera de las asignadas. La restricción es estructural y no depende del criterio del modelo en tiempo de ejecución, lo que la hace predecible y auditable.

E5. Stateful Runner. Pasar una sesión persistente al Runner del SDK permite mantener el historial de conversación entre peticiones sin código adicional de serialización. La ventaja es que el SDK gestiona la consistencia del estado entre turnos; la desventaja es que la sesión queda atada al proceso servidor, lo que limita la escalabilidad horizontal.

E6. Quality by Evaluation. Ejecutar una batería de casos de prueba con evaluadores LLM especializados por categoría de riesgo permite detectar problemas de calidad que los tests de integración convencionales no cubren: alucinaciones, sesgo, desinformación, actualidad y vulnerabilidad a jailbreak. Cada evaluador produce una salida estructurada con puntaje, indicación booleana y justificación, persistida con timestamp para trazabilidad longitudinal.

E7. Quota-Driven Degradation. Calcular el modelo efectivo antes de cada petición en función del consumo diario del usuario permite degradar automáticamente a un modelo más económico cuando se supera el límite, manteniendo la continuidad del servicio. La degradación es transparente para el usuario salvo por un aviso explícito.

E8. MCP Tool Filter. Filtrar el conjunto de herramientas expuestas por un servidor MCP antes de registrarlas en el agente reduce la superficie de ataque. El filtro, implementado como función en el constructor del servidor, bloquea explícitamente herramientas con privilegios elevados antes de que el LLM pueda invocarlas.

E9. Per-User Vectorstore. Asignar a cada usuario su propio índice FAISS en disco garantiza que un usuario no pueda recuperar accidentalmente documentos de otro. El índice se cachea en memoria entre peticiones del mismo usuario y se invalida cuando se modifica el contenido del sandbox.

E10. Deterministic Intent Classification. Clasificar la entrada del usuario mediante expresiones regulares y un sistema de puntuación de especificidad antes de invocar el LLM permite descartar consultas triviales sin incurrir en el costo de una llamada al modelo. En el sistema donde se implementó, el equipo estimó una reducción de llamadas innecesarias a la cadena RAG en consultas triviales, aunque esta cifra no fue medida de forma sistemática.

Listing 2: Pseudocódigo abstracto: Deterministic Intent Classification (E10).

```
Entrada: texto t, umbral_puntos U
Salida:  decision {TRIVIAL, SUSTANTIVA}

si patron_saludo(t):       // expresion regular
    retornar TRIVIAL

puntos <- 0
para cada indicador en {tecnologia, anios_exp, seniority,
                        formacion, certificacion}:
    si indicador presente en t:
        puntos <- puntos + 1

si puntos >= U:            // U = 3 en la implementacion
    retornar SUSTANTIVA
retornar TRIVIAL
```

La Sección 5 describe las implementaciones concretas de estas estrategias en cada sistema, con los detalles técnicos y las decisiones específicas para cada contexto.

5. Casos de estudio

5.1. NeuroHire

NeuroHire es el primer sistema de esta familia. Automatiza tres etapas del proceso de selección de personal: gestión centralizada de candidatos y currículum, búsqueda semántica en lenguaje natural y entrevistas laborales guiadas por inteligencia artificial

Tabla 2: Presencia de estrategias de diseño por sistema.

Estrategia	NeuroHire	Doki	MateApp	Ciruelito
E1 Agent as Tool	—	▲	▲	▲
E2 SandboxPath Injection	—	▲	—	—
E3 Threshold RAG	▲	▲	▲	▲
E4 Role-Based Tooling	—	▲	▲	—
E5 Stateful Runner	—	—	—	▲
E6 Quality by Evaluation	—	—	—	▲
E7 Quota-Driven Degradation	—	▲	—	—
E8 MCP Tool Filter	—	▲	—	—
E9 Per-User Vectorstore	△	▲	—	—
E10 Deterministic Intent	▲	—	—	—

▲ = implementada; △ = parcial/prevista; — = no aplica.

con análisis de video y voz. Está orientado a organizaciones de tamaño mediano y grande con volúmenes de candidatos que hacen inviable la revisión manual. Adopta una arquitectura de microservicios con orquestación vía API REST, con la orquestación del RAG implementada en LangChain 0.0.350 y el módulo de entrevistas con lógica directa sobre el SDK de OpenAI en TypeScript.

El agente RAG utiliza gpt-4 con temperatura 0 y max_tokens 1000, con modelo de embeddings text-embedding-ada-002 y parámetros de particionado chunk_size 800 y un overlap de chunks 100. El agente Entrevistador utiliza como modelo gpt-4.1-mini con temperatura variable: 0.7 para la generación de preguntas, 0.3 para el análisis de CV (max_tokens 1500) y 0.2 para la transcripción con Whisper-1 y el análisis de video (max_tokens 2000).

NeuroHire implementa un recuperador personalizado ThresholdRetriever (E3) sobre FAISS que convierte distancia a similitud coseno antes de aplicar el umbral:

Listing 3: ThresholdRetriever: conversión distancia–similitud (NeuroHire).

```
class ThresholdRetriever(BaseRetriever):
    def __init__(self, vectorstore, threshold=0.7, k=10):
        super().__init__()
        self.vectorstore = vectorstore
        self.threshold = threshold  # 0.7: filtra contexto de baja confianza
        self.k = k

    def _get_relevant_documents(self, query: str, *, run_manager) -> List[Document]:
        docs_with_scores = self.vectorstore.similarity_search_with_score(query, k=self.k)
        filtered_docs = []
        for doc, distance in docs_with_scores:
            similarity = 1 - distance  # FAISS devuelve distancias, no similitudes
            if similarity >= self.threshold:
                filtered_docs.append(doc)
```

```
        return filtered_docs
```

El patron Strategy funciona con distintas estrategias, el patró State funciona por tipo: el procesamiento de documentos sigue un patrón Strategy alternando estrategias según MIME: PDF con PyMuPDF, DOCX con python-docx e imágenes con Tesseract OCR bilingüe. Existe una implementación paralela en TypeScript resultado de la independencia operativa de los equipos, lo que implica mantener dos implementaciones en paralelo. La clasificación determinista de intención (E10) asigna puntos por presencia de indicadores concretos como tecnologías nombradas, años de experiencia y nivel de senior, y ejecuta la cadena RAG solo si la entrada acumula al menos tres indicadores. El primer análisis de un CV se persiste en PostgreSQL y las llamadas posteriores reutilizan el resultado. La evaluación del nivel de inglés solo se activa si Whisper detecta inglés como idioma de la respuesta, evitando penalizar a candidatos que se presentan en español.

5.2. Doki

Doki es un asistente inteligente para el manejo de documentos por usuario y por empresa. Combina RAG, búsqueda web bajo demanda, visualización de gráficas, acceso a PDFs remotos, generación de imágenes y resúmenes exportables en PDF, PNG y JSON. Su modelo de negocio es multi-tenant con jerarquía superadmin/admin/cliente/usuario y un límite de tres clientes por empresa.

El QA Agent actúa como orquestador centralizado y delega en subagentes RAG y Web mediante agent.as_tool. El modelo efectivo se calcula en tiempo de ejecución según las cuotas del usuario y soporta desde gpt-5.2 hasta gpt-4.1-mini. Los subagentes utilizan gpt-4.1-mini y los embeddings emplean text-embedding-3-small con $k = 8$.

Para garantizar que la información de cada usuario no se cruce entre sesiones, Doki implementa SandboxPath Injection (E2) mediante la variable `ContextVar`:

Listing 4: ContextVar para corrección de sandbox (Doki).

```
_current_request_sandbox_path: ContextVar[Optional[str]] = ContextVar(
    "current_request_sandbox_path", default=None)

@function_tool
def rag_tool_function(question: str, sandboxpath: str) -> str:
    """Consulta sobre la question en los documentos del usuario."""
    effective_path = get_current_request_sandbox_path() or sandboxpath
    if get_current_request_sandbox_path():
        _log.info("Usando sandbox inyectado (ignorando param del modelo)
    : %s", effective_path)
    rag_tool = tai.RAGTool(folder_path=effective_path, ...)
    return rag_tool.retrieve_context(question)
```

Para el Quota-Driven Degradation (E7), el reinicio de cuota se produce a las 13:00 hora argentina en lugar de a medianoche, distribuyendo la carga y evitando el pico de actividad nocturno característico de servicios con base de usuarios local:

Listing 5: Reinicio de cuota a las 13:00 Argentina (Doki, SQL).

```
CREATE OR REPLACE FUNCTION get_reset_date_argentina() RETURNS DATE AS $$
DECLARE
```

```
    v_now_arg TIMESTAMP;
    v_hour    INTEGER;
BEGIN
    v_now_arg := (NOW() AT TIME ZONE 'UTC') AT TIME ZONE 'America/
    Argentina/Buenos_Aires';
    v_hour    := EXTRACT(HOUR FROM v_now_arg);
    IF v_hour >= 13 THEN
        RETURN (v_now_arg::DATE + INTERVAL '1 day')::DATE;
    ELSE
        RETURN v_now_arg::DATE;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

En cuanto a seguridad, Doki implementa protección contra SSRF resolviendo las IPs del hostname y bloqueando rangos privados o reservados (RFC 1918, loopback, link-local e IPv6 privado) antes de ejecutar cualquier petición HTTP, para mitigar ataques de DNS rebinding en entornos multi-tenant.

5.3. MateApp

MateApp es una plataforma experimental desarrollada en el marco de actividades de I+D que asiste a la comunidad académica mediante dos roles diferenciados: consultas administrativas para profesores, materias, horarios e inscripciones, y asistencia pedagógica con contenidos de cursos y material de estudio. Dos endpoints de chat (`/chat/admin`, `/chat/profesor`) instancian agentes distintos en el arranque de FastAPI, compartiendo cuatro servidores MCP: fetch, Playwright, filesystem y Firecrawl. Los cuatro agentes utilizan gpt-4o-mini con embeddings text-embedding-3-small y $k = 4$ sobre un índice FAISS compartido y persistente.

MateApp implementa Role-Based Tooling (E4) con una separación de herramientas por rol que no puede ser eludida por el LLM:

Listing 6: Separación de roles por lista de herramientas (MateApp).

```
agente_administrativo = Agent(
    name="Administrativo",
    instructions=ins.instructions_admin,
    tools=[asdk.postgres_get_tables, asdk.postgres_get_all, asdk.
    postgres_update],
    mcp_servers=[mcp_fetch, mcp_playwright, mcp_filesystem,
    mcp_firecrawl],
    model="gpt-4o-mini")

agente_profesor = Agent(
    name="Profesor",
    instructions=ins.instructions_profe,
    tools=[asdk.postgres_get_tables, asdk.postgres_get_all, asdk.
    postgres_update,
           asdk.rag_search, asdk.web_search, asdk.download_files],
    mcp_servers=[mcp_fetch, mcp_playwright, mcp_filesystem,
    mcp_firecrawl],
    model="gpt-4o-mini")
```

5.4. Ciruelito

Ciruelito es una plataforma experimental desarrollada en el marco de actividades de I+D que actúa como asistente de investigación con tres fuentes de conocimiento: documentos propios del usuario, web general y datos de proyectos en Azure DevOps. El agente orquestador Research Agent utiliza gpt-4o-mini y expone tres subagentes: rag_search con el modelo gpt-5-nano, web_search usando gpt-4o-mini y el parámetro tool_choice=required", y azdo_search con catorce (14) herramientas asincrónicas sobre la API REST de Azure DevOps.

El RAG es de tipo multimodal con embeddings text-embedding-3-small y $k = 4$. Los documentos se procesan con PyMuPDF, las imágenes extraídas se describen con gpt-5-nano (poderoso y bajo en costos) y OCR Tesseract para extraer información, y las descripciones se concatenan al texto de página antes de indexar. Un caché por hash SHA-1 evita reprocesar imágenes ya vistas y el procesamiento paralelo con ThreadPoolExecutor reduce la latencia de ingesta.

6. Resultados

Se logró implementar en entornos de desarrollo los cuatro sistemas de inteligencia agentica para escenarios de uso reales: reclutamiento y entrevistas asistidas con análisis de video y audio (NeuroHire), asistencia documental multi-tenant en tiempo real bajo restricciones de costo (Doki), asistente administrativo y pedagógico para la comunidad académica (MateApp), e investigación con fuentes híbridas y gestión de proyectos (Ciruelito)

Entre las estrategias con mayor impacto observado se encuentran las siguientes. Threshold RAG (E3) resolvió el problema de que FAISS devuelve los k vecinos más cercanos con independencia de su relevancia absoluta: sin umbral, el modelo generaba respuestas basadas en documentos de baja similitud. Los valores de umbral observados fueron 0.7 en similitud coseno (NeuroHire), 0.5 en score normalizado (Ciruelito) y 1.5 en distancia L2 máxima (Doki). SandboxPath Injection (E2) surgió de un comportamiento observado en el desarrollo de Doki, donde el LLM pasaba el nombre de un archivo como sandboxpath en lugar de la ruta del directorio del usuario, comportamiento impredecible desde el prompt y difícil de depurar por su carácter intermitente. Quota-Driven Degradation (E7) mantuvo la continuidad del servicio sin interrupciones mediante un reinicio de cuota a las 13:00 para evitar el pico de actividad nocturno. Quality by Evaluation (E6) en Ciruelito aportó un enfoque de evaluación formal que incluyó casos como la evasión de seguridad de base de datos mediante jailbreak. Deterministic Intent Classification (E10) en NeuroHire permitió filtrar consultas triviales mediante expresiones regulares, reduciendo el costo de llamadas innecesarias al LLM.

Las estrategias restantes (E1, E4, E5, E8, E9) figuran en la tabla 2 y se describen en las secciones específicas de cada sistema; su omisión en esta discusión responde a que sus implementaciones siguen patrones ya establecidos en la literatura y no generaron hallazgos adicionales relevantes.

7. Discusión

NeuroHire adoptó un modelo de microservicios heterogéneos sin implementación del SDK, donde la orquestación la realizan el frontend y los backends vía API REST. Doki, MateApp y Ciruelito comparten el patrón de un único orquestador por petición HTTP con delegación a subagentes como herramientas. Las diferencias entre estos tres sistemas son las siguientes: Ciruelito incorporó memoria de sesión nativa del SDK mediante SQLiteSession, Doki añadió streaming SSE, cuotas por usuario y vectorstore aislado por usuario, y MateApp separó roles por endpoints dedicados con instrucciones y herramientas distintas por agente.

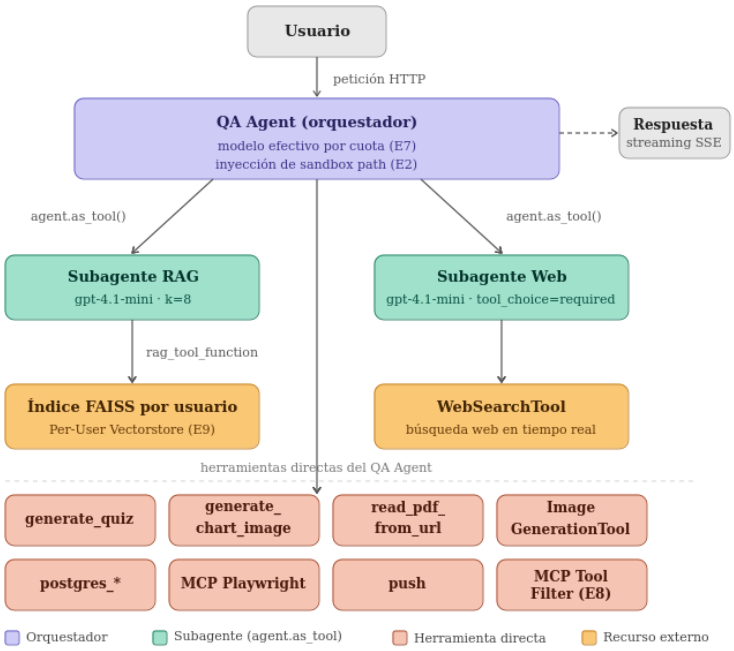

Figura 2: Topología de agentes del sistema Doki. El QA Agent actúa como orquestador centralizado y delega en subagentes mediante `agent.as_tool()`, mientras que las herramientas directas operan bajo su control sin transferencia de control.

En los sistemas analizados se observó una convergencia hacia el patrón de orquestador único con delegación mediante agent.as_tool. Ninguno de los cuatro sistemas implementó handoffs en desarrollo debido a que agent.as_tool produce un resultado textual observable más fácil de depurar, mientras que los handoffs realizan un cambio de control por el cual el agente destino accede a la sesión del usuario, introduciendo complejidad operativa dado que la memoria del agente raíz no incluye lo ocurrido durante la

delegación.

Las estrategias presentes en los sistemas muestran un conjunto de prácticas que son transversales, como son: utilizar RAG con FAISS y umbral de confianza, subagentes expuestos mediante agent.as_tool, restricción de herramientas SQL por agente, instrucciones en módulos separados, trazabilidad con Trace y llamados asíncronos con AsyncExitStack para MCP. La tabla 3 resume las dimensiones clave de los cuatro sistemas.

Tabla 3: Comparativa de los cuatro sistemas.

Proyecto	**Arquitectura**	**# Ag.**	**# Tools**	**Estrategias clave**
NeuroHire	Microservicios REST Python+TS, sin SDK	2	8+	E3, E10, caché CV, SSO, Whisper
Doki	SDK: QA + 2 sub-ag., SSE, multi-tenant	3	15+	E1–E4, E7–E9, SSRF, pool PG
MateApp	SDK: 2 princip. + 2 sub-ag., MCP	4	6+	E1, E4, FAISS compartido, difflib
Ciruelito	SDK: 1 + 3 sub-ag., AzDO, sesión	4	20+	E1, E3, E5, E6, multimodal, PAT

8. Conclusiones

Este trabajo documentó cuatro sistemas multiagente en Python, tres basados en el OpenAI Agents SDK y uno en LangChain y TypeScript, como familia de casos de estudio. El análisis revela un núcleo de estrategias compartidas: delegación jerárquica, umbral RAG, lista cerrada de herramientas, módulo de instrucciones separado, trace y AsyncExitStack, que se combinan con estrategias específicas de dominio como cuotas, vectorstore por usuario, integración con Azure DevOps y evaluación formal. La elección entre los mecanismos de delegación sigue una distinción observada en los proyectos: @function_tool para lógica determinista, agent.as_tool para subtareas con razonamiento propio, y handoffs para cambios de control de la conversación. La ausencia de handoffs en los cuatro sistemas es el hallazgo más relevante del corpus: en aplicaciones orientadas a preguntas y respuestas, agent.as_tool cubrió la mayoría de los escenarios de delegación entre agentes.

En cuanto a la posibilidad de trasladar estas observaciones a otros contextos, algunas estrategias responden a problemas estructurales del uso de LLMs con herramientas que podrían aparecer en dominios distintos a los analizados. Threshold RAG (E3) aborda una limitación técnica de FAISS que es independiente del dominio de aplicación. Role-Based Tooling (E4) y MCP Tool Filter (E8) responden a consideraciones de seguridad presentes en cualquier sistema que exponga herramientas con privilegios distintos. Agent as Tool (E1) y Deterministic Intent Classification (E10) surgen de restricciones operativas del SDK y del costo de inferencia que son transversales a múltiples casos de uso.

Otras estrategias están más acotadas al contexto donde surgieron. Quota-Driven Degradation (E7) con reinicio a las 13:00 hora argentina es una respuesta específica al perfil de uso de Doki. SandboxPath Injection (E2) surgió de un comportamiento particular del modelo en producción. En ambos casos, el principio subyacente, guardrails imperativos ante comportamientos impredecibles del LLM, podría ser útil como referencia en otros sistemas, aunque la implementación concreta dependería del contexto.

8.1. Aclaración

Durante la preparación de este trabajo, los autores utilizaron Claude Sonnet4.6 con el fin de mejorar el lenguaje y la legibilidad del manuscrito. Tras utilizar esta herramienta de IA generativa, los autores revisaron y editaron el contenido según fuera necesario y asumen plena responsabilidad por el contenido de la publicación.

Referencias

Althaf, A. M., Mohammed, M. A., Talburt, J. R., and Cakmak, M. C. (2025). Multi-agent RAG framework for entity resolution: Advancing beyond single-LLM approaches with specialized agent coordination. *Computers*, 14(12):525.

Anthropic et al. (2024). Model context protocol specification. https://modelcontextprotocol.io/.

Guo, T., Chen, X., Wang, Y., Chang, R., Pei, S., Chawla, N. V., Wiest, O., and Zhang, X. (2024). Large language model based multi-agents: A survey of progress and challenges. pages 8048–8057.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Wang, J., Zhang, C., Wang, Z., Yau, S. K. S., Lin, Z., et al. (2024). Metagpt: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

Johnson, J., Douze, M., and Jégou, H. (2019). Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Li, X., Wang, S., Zeng, S., Wu, Y., and Yang, Y. (2024). A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(9).

OpenAI (2025). OpenAI Agents SDK: Build agentic applications with minimal abstraction. https://openai.github.io/openai-agents-python/. Documentación técnica oficial.

Russell, S. J. and Norvig, P. (2004). Inteligencia artificial: un enfoque moderno.

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2024). Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. (2024). A prompt pattern catalog to enhance prompt engineering with ChatGPT. In *Companion of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 56–67.

Wooldridge, M. (2009). *An Introduction to Multiagent Systems*. John Wiley & Sons, 2 edition.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. (2023). Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*.

Xi, Z., Chen, W., Guo, X., He, W., Ding, Y., Hong, B., Zhang, M., Wang, J., Jin, S., Zhou, E., et al. (2025). The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*.

Xie, Y., Jiang, B., Mallick, T., Bergerson, J., Hutchison, J. K., Verner, D. R., et al. (2025). Marsha: multi-agent rag system for hazard adaptation. *npj Climate Action*, 4(70).

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023). ReAct: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Yin, R. K. (2018). *Case Study Research and Applications: Design and Methods*. SAGE Publications, 6 edition.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E. P., et al. (2023). Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, volume 36.