FACULTAD DE INGENIERÍA

TRABAJO PRÁCTICO FINAL

PROCESAMIENTO DEL LENGUAJE NATURAL III

MAESTRÍA EN INTELIGENCIA ARTIFICIAL

Universidad de Buenos Aires (FIUBA)

UI2Code

Sistema Multi-Agente para conversión de diseños UI a código HTML/Tailwind CSS y prompt a código

---

Equipo:

Noelia Melina Qualindi	noelia.qualindi@gmail.com
Fabricio Denardi	denardifabricio@gmail.com
Jorge Ceferino Valdez	jorgecvaldez@gmail.com
Bruno Masoller	brunomaso1@gmail.com

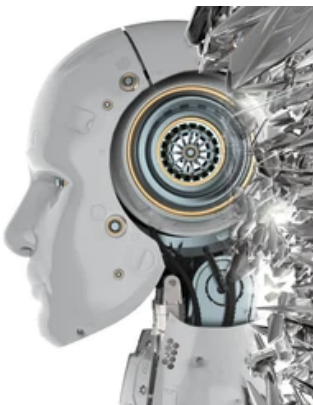
Docentes:

Oksana Bokhonok, Abraham Rodríguez

Repositorio del proyecto:

https://github.com/jorgeceferinovaldez/ui2code-rag

9 de octubre de 2025



Índice

1. Resumen

UI2Code-RAG es un sistema multi-agente que convierte diseños de interfaz de usuario (imágenes estáticas) en código HTML + Tailwind CSS funcional y estandarizado mediante la combinación de análisis visual con Inteligencia Artificial y RAG híbrido (BM25 + embeddings vectoriales + re-ranking con cross-encoder) sobre un corpus de 900 ejemplos HTML/CSS del dataset WebSight.

La arquitectura se orquesta mediante un **Orchestrator Agent A2A** que coordina tres agentes especializados:

- **Visual Agent**: Análisis multimodal de imágenes UI
- **RAG Agent**: Búsqueda híbrida de patrones de código similares
- **Code Agent**: Generación de código HTML/Tailwind con validación anti-hallucination

El sistema incluye guardrails de validación de salidas (JSON, HTML, web sanitization) y una aplicación Streamlit multipágina que proporciona interfaces para: UI-to-Code, Query Interface, Evaluaciones, System Status y Corpus Information.

2. Presentación del problema

En ingeniería de software, uno de los principales problemas para las empresas que desarrollan grandes sistemas de información es el balance entre **estándares flexibles y estándares forzosos**. La estandarización de las prácticas de ingeniería de software siempre es mejor, usualmente sin importar el caso de uso, pero esto lleva a un mayor "lead time". En un mundo actual donde el "time to market" es uno de los principales atributos dentro de un proceso, este complejo problema lleva a soluciones del tipo *"convention over configuration"*.

El potencial de los LLMs (Large Language Models) los hace aptos para este tipo de desarrollo, donde se puede forzar la convención de forma transparente para el desarrollador.

2.1. Contexto y motivación

Con este tema en mente, **UI2Code** se posiciona con la misión de brindar a los desarrolladores **estandarización automática**, al generar código que cumpla con los estándares de la empresa en base a un diseño preliminar. Por un lado, los desarrolladores pueden partir de una base estandarizada y, por el otro, se reducen los tiempos de desarrollo para las empresas.

2.2. Desafíos identificados

Sin embargo, existen varios desafíos técnicos:

1. **Alucinación de modelos**: Los modelos de visión-lenguaje (VLMs) y LLMs pueden alucinar estructura o estilos inexistentes cuando deben generar código a partir de una sola imagen.

2. **Ambigüedad semántica**: Distintas UIs presentan ambigüedad en layout, jerarquías, tipografías y breakpoints responsivos.
3. **Falta de evidencia explícita**: Ciertos componentes no tienen evidencia clara en la imagen para inferir su implementación técnica.
4. **Necesidad de contexto reutilizable**: Se requiere un corpus de patrones de código que guíe la generación y mantenga la consistencia.
5. **Verificación de formato**: Es necesario implementar validaciones para reducir alucinaciones y obtener código realmente utilizable.

3. Objetivos

3.1. Objetivo general

Desarrollar un sistema multi-agente que convierta imágenes de interfaces de usuario en código HTML/Tailwind CSS "listo para usar" para minimizar alucinaciones y maximizar la reutilización de patrones estandarizados.

3.2. Objetivos específicos

- **Conversión UI-to-Code**: Transformar imágenes de diseños UI en código HTML/-Tailwind funcional y estandarizado.
- **Recuperación y reutilización de patrones**: Implementar un sistema RAG híbrido para recuperar y reusar patrones de código similares con el fin de guiar la generación y reducir la deuda técnica.
- **Interfaz de usuario simplificada**: Exponer una interfaz web intuitiva para cargar imágenes, ajustar instrucciones personalizadas, visualizar resultados y descargar código generado.
- **Visibilidad del sistema**: Proporcionar dashboards de estado del sistema, métricas del corpus y health checks de los agentes.
- **Evaluación cuantitativa**: Implementar framework de evaluación con métricas estándar de recuperación de información.

3.3. Alcance

- Inferencia mediante modelos de IA vía API (sin entrenamiento)
- Corpus curado de 900 patrones HTML/CSS del dataset WebSight
- Sistema RAG híbrido con BM25, búsqueda vectorial y re-ranking
- Validaciones de salida con Guardrails
- Arquitectura multi-agente con protocolo A2A (Agent-to-Agent)

4. Técnicas y tecnologías utilizadas

4.1. Modelos de Inteligencia Artificial

4.1.1. Análisis visual

- **Gemini 2.0 Flash Exp** (Google): Modelo de visión gratuito para análisis de layout, componentes y estilos
- **GPT-4 Vision** (OpenAI): Alternativa de pago con mayor precisión
- **Claude 3.5 Sonnet** (Anthropic): Opción premium para casos complejos

4.1.2. Generación de código

- **DeepSeek R1 Distill Llama 70B**: Modelo gratuito optimizado para código
- **DeepSeek R1**: Versión de pago con mejor calidad
- Acceso vía **OpenRouter**: Plataforma que unifica el acceso a múltiples modelos con precios competitivos

4.2. Sistema RAG híbrido

El sistema implementa un pipeline de recuperación de información de tres capas:

1. **Búsqueda léxica (BM25)**: Recuperación basada en términos exactos y frecuencia
2. **Búsqueda vectorial (Semantic Search)**:
   - Embeddings con `sentence-transformers/all-MiniLM-L6-v2` (384 dimensiones)
   - Almacenamiento en Pinecone Vector Database
   - Namespace: `html-css-examples`
3. **Re-ranking con Cross-Encoder**:
   - Modelo: `ms-marco-MiniLM-L-6-v2`
   - Mejora la precisión del ranking final

4.3. Infraestructura y orquestación

4.3.1. Protocolo A2A (Agent-to-Agent)

- Comunicación basada en JSONRPC sobre HTTP
- Endpoints estandarizados: `/.well-known/agent-card.json`
- SDK: `a2a-sdk >= 0.3.8`
- Servidor ASGI: Uvicorn + Starlette

4.3.2. Validación y guardrails

- Framework: `guardrails-ai >= 0.6.7`
- Validadores utilizados:
  - `valid_json`: Formato JSON válido
  - `regex_match`: Patrones específicos
  - `web_sanitization`: Limpieza de HTML inseguro
  - `valid_schema_json`: Schema Pydantic (custom)
  - `is_html_field`: Validación HTML con BeautifulSoup (custom)

4.3.3. Aplicación web y UI

- Framework: `Streamlit >= 1.28.0`
- Puerto: 8501
- Arquitectura multipágina con navegación lateral
- Componentes interactivos: drag&drop, preview HTML, download buttons

4.3.4. Contenerización

- Docker + Docker Compose para orquestación
- Tres contenedores principales:
  - Visual Agent (puerto 10000)
  - Code Agent (puerto 10001)
  - Streamlit App (puerto 8501)
- Script automatizado: `run.sh` para configuración y despliegue
- Portable a Kubernetes con herramientas como Kompose

4.4. Datos y corpus

4.4.1. Dataset WebSight

- Fuente: HuggingFace (`HuggingFaceM4/WebSight`)
- Versión utilizada: v0.2
- Documentos: 900 ejemplos HTML/CSS
- Campos utilizados:
  - `text`: Código HTML completo
  - `llm_generated_idea`: Descripción del componente (para evaluación)
- Almacenamiento local: `data/websight/*.json`
- Chunking: max_tokens=400, overlap=100

4.4.2. Dataset de evaluación

- Ubicación: `data/evaluate/`
- Documentos: 9 ejemplos HTML con descripciones UI
- Qrels: Labels de relevancia query-documento
- Formato: JSONL + CSV

4.5. Logging y monitoreo

- Sistema de logging: `loguru >= 0.7.3`
- Logs estructurados en `logs/ui2code.YYYY-MM-DD.log`
- Rotación automática por fecha
- Niveles configurables (DEBUG, INFO, WARNING, ERROR)

5. Arquitectura del sistema

5.1. Visión general

El sistema UI2Code implementa una arquitectura multi-agente coordinada mediante el protocolo A2A (Agent-to-Agent), donde agentes especializados trabajan en conjunto para resolver el problema de conversión UI-to-Code.

5.2. Pipeline lógico (alto nivel)

El flujo completo del sistema sigue estos pasos:

1. **Input**: Usuario proporciona imagen UI + instrucciones personalizadas
2. **Agente Visual (A2A)**: Realiza análisis multimodal de la imagen
   - Identifica componentes (botones, formularios, cards, etc.)
   - Analiza layout (grid/flex/columnas, jerarquía)
   - Extrae estilo (paleta de colores, tipografía, spacing)
   - Genera análisis estructurado en JSON
3. **RAG Agent**: Recupera patrones HTML/CSS similares del corpus
   - Búsqueda híbrida: BM25 + Vector Search
   - Re-ranking con Cross-Encoder
   - Enriquecimiento con código HTML completo (hasta 4484 chars)
   - Retorna top-k patrones más relevantes
4. **Agente de Código (A2A)**: Genera código HTML/Tailwind
   - Condiciona generación con: análisis visual + patrones RAG + instrucciones

- Formatea patrones (primeros 2000 caracteres)
- Aplica prompts optimizados con reglas anti-hallucination
- Valida componentes generados vs componentes solicitados

5. **Guardrails**: Validan esquema y sanitizan contenido
   - Verificación de JSON válido y schema Pydantic
   - Validación de HTML con BeautifulSoup
   - Web sanitization (evitar código malicioso)
6. **Output**: Código HTML + Tailwind CSS funcional
   - Sin dependencias de icon libraries externas
   - Diseño artesanal y limpio
   - Metadata de generación incluida

5.3. Componentes principales

5.3.1. Visual agent (puerto 10000)

- **Tecnología**: Agente A2A con modelos de visión-lenguaje
- **Modelos**: Gemini 2.0 Flash / GPT-4 Vision / Claude 3.5 Sonnet
- **Configuración**: Lee `VISUAL_MODEL` desde `src/agents/visual_a2a_agent/.env`
- **Endpoint A2A**: `/.well-known/agent-card.json`
- **Salida**: JSON estructurado con componentes, layout, estilo, color_scheme

5.3.2. RAG agent (módulo Python)

- **Tecnología**: Agente determinista, sin servidor HTTP
- **Función**: Orquesta pipeline de recuperación híbrida
- **Componentes**:
  - `WebSightLoader`: Carga 900 documentos de `data/websight/`
  - `BM25Search`: Búsqueda léxica
  - `PineconeSearcher`: Búsqueda vectorial semántica
  - `CrossEncoder`: Re-ranking de candidatos
- **Salida**: Lista de tuplas `(doc_id, chunk, metadata, score)`
- **Enriquecimiento**: Agrega `html_code` completo a metadata

5.3.3. Code agent (puerto 10001)

- **Tecnología**: Agente A2A con modelos de código
- **Modelos**: DeepSeek R1 70B / GPT-4 / Claude 3.5
- **Configuración**: Lee `CODE_MODEL` desde `src/agents/code_a2a_agent/.env`
- **Endpoint A2A**: `/.well-known/agent-card.json`
- **Validación anti-hallucination**:
  - Método `_validate_html_components()`
  - Detecta secciones extra (header, nav, footer, aside)
  - Compara con componentes del análisis visual
  - Logging de warnings si hay discrepancias
- **Salida**: JSON con `html_code` y `generation_metadata`

5.3.4. Orchestrator agent

- **Tecnología**: Agente determinista coordinador
- **Función**: Orquesta el flujo completo entre agentes
- **Protocolo**: JSONRPC sobre HTTP
- **Responsabilidades**:
  - Inicializar conexiones con Visual, RAG y Code Agents
  - Fetch de agent cards vía `A2ACardResolver`
  - Crear `A2AClient` para cada agente
  - Manejar errores y validar respuestas
  - Consolidar resultados finales

5.4. Diagrama de flujo completo

El sistema sigue el flujo ilustrado en la Figura 1, donde:

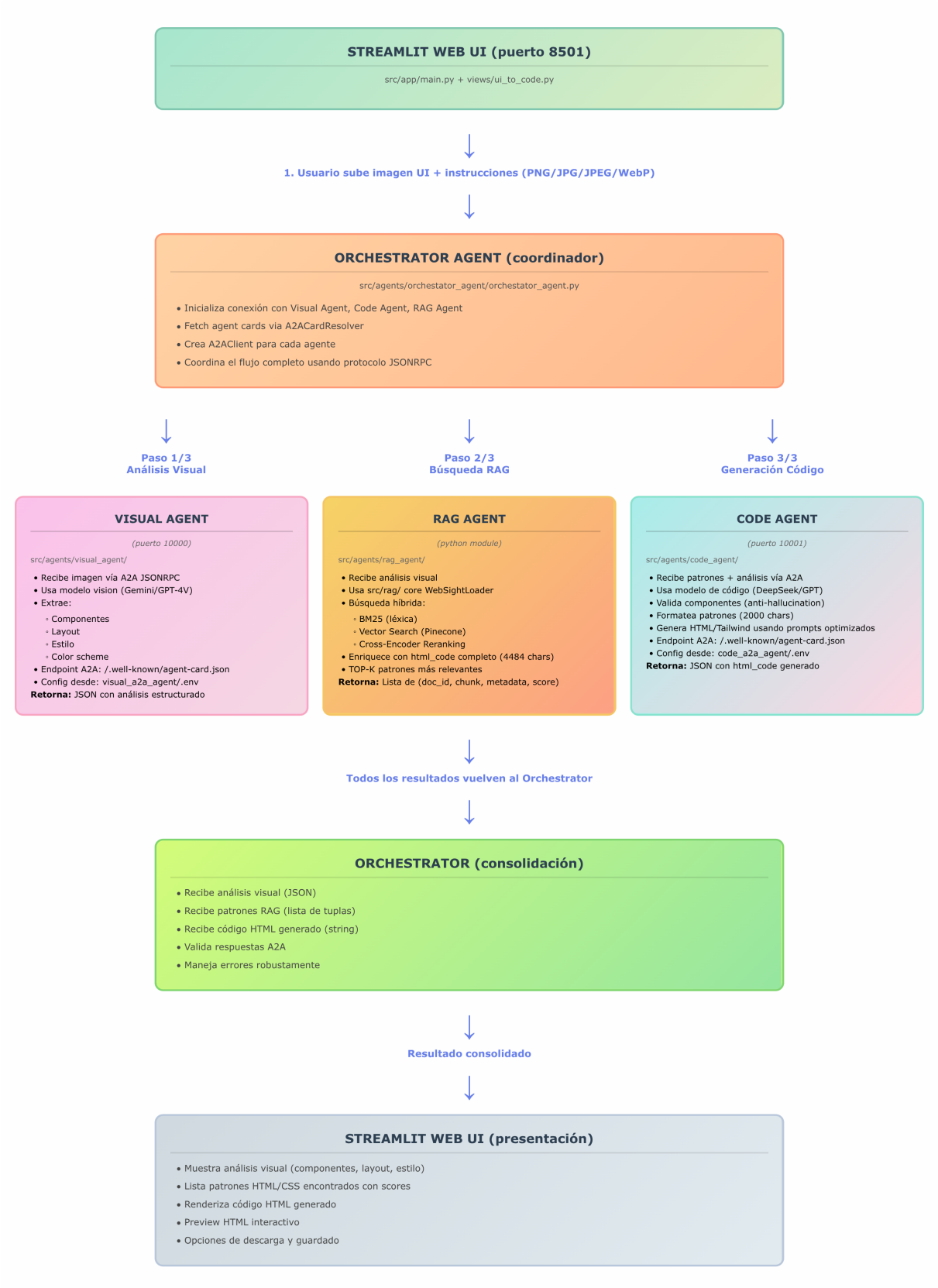


Figura 1: Arquitectura Multi-Agente UI2Code

1. Usuario sube imagen UI + instrucciones en Streamlit (puerto 8501)

2. Orchestrator Agent coordina la ejecución secuencial:
   - **Paso 1/3**: Visual Agent analiza imagen vía A2A JSONRPC
   - **Paso 2/3**: RAG Agent busca patrones similares
   - **Paso 3/3**: Code Agent genera HTML usando análisis + patrones
3. Orchestrator consolida resultados y retorna a Streamlit
4. Streamlit presenta: análisis visual, patrones encontrados, código generado, preview HTML

5.5. Pipeline RAG detallado

El sistema RAG implementa un flujo de tres fases como se muestra en la Figura 2:

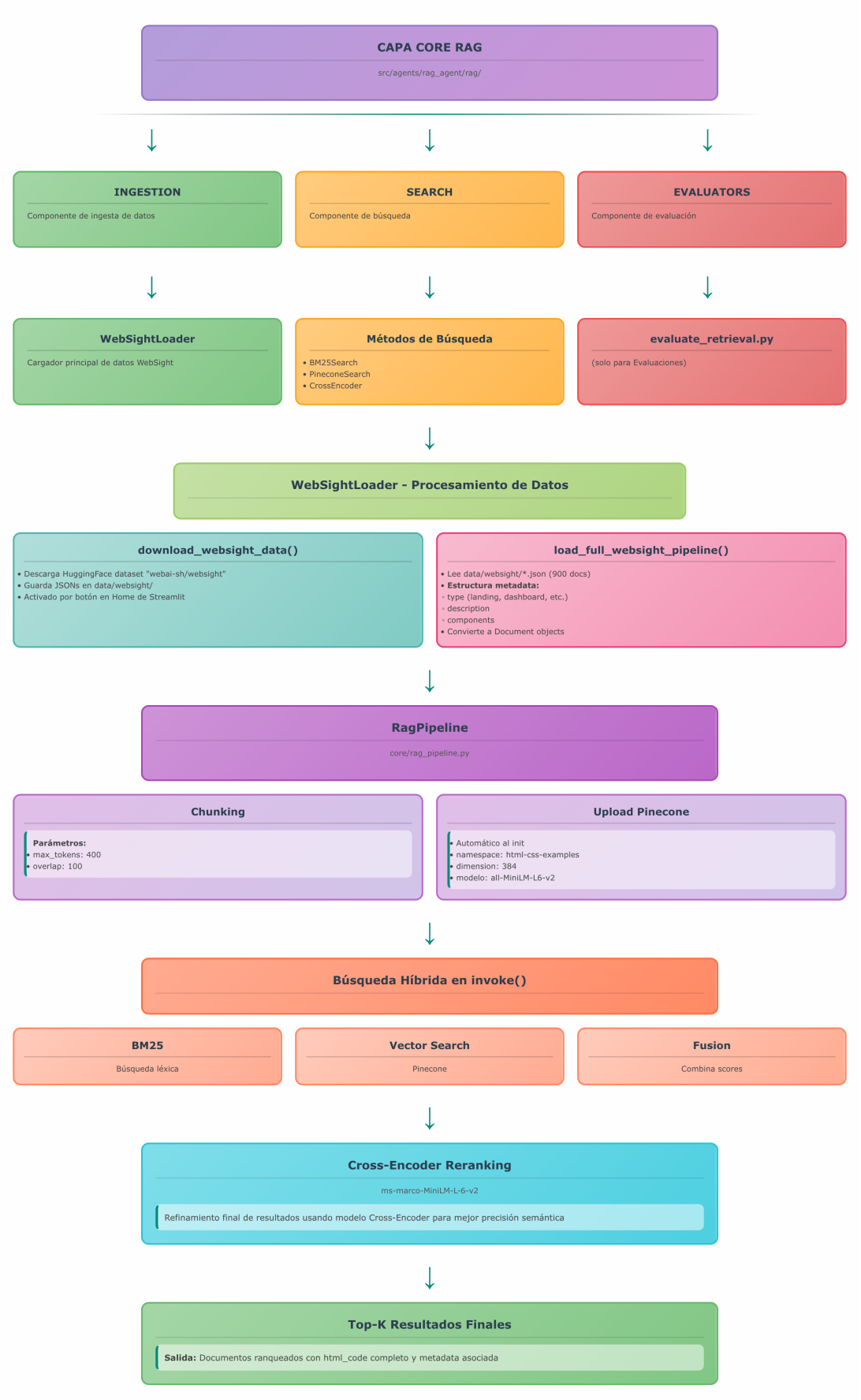


Figura 2: Pipeline RAG Híbrido

5.5.1. Fase 1: Ingesta de documentos

- `WebSightLoader.download_websight_data()`: Descarga dataset desde HuggingFace
- Almacenamiento en `data/websight/*.json`
- `load_full_websight_pipeline()`: Carga 900 documentos con metadata estructurado
- Conversión a objetos `Document` con campos: type, description, components

5.5.2. Fase 2: Procesamiento

- **Chunking**: max_tokens=400, overlap=100
- **Embedding**: `all-MiniLM-L6-v2` (384 dimensiones)
- **Upload a Pinecone**: Namespace `html-css-examples`
- Proceso automático al inicializar `RagPipeline`

5.5.3. Fase 3: Búsqueda híbrida (método `invoke()`)

1. **BM25 Search**: Recuperación léxica de candidatos
2. **Vector Search**: Consulta en Pinecone (cosine similarity)
3. **Fusion**: Combina scores de ambos métodos con pesos configurables
4. **Cross-Encoder Re-ranking**: Ordena candidatos por relevancia real
5. **Enriquecimiento**: Agrega `html_code` completo a metadata (hasta 4484 chars)
6. **Top-k final**: Retorna los k resultados más relevantes

5.6. Protocolos de comunicación

Cuadro 1: Protocolos de comunicación entre componentes

Comunicación	**Protocolo**
Streamlit ↔ Orchestrator	Python directo (`asyncio`)
Orchestrator ↔ Visual Agent	A2A JSONRPC / HTTP (puerto 10000)
Orchestrator ↔ Code Agent	A2A JSONRPC / HTTP (puerto 10001)
Orchestrator ↔ RAG Agent	Python directo (módulo local)
RAG Agent ↔ Pinecone	gRPC (Pinecone API)

5.7. Estructura de directorios

La estructura del proyecto está organizada modularmente:

Listing 1: Estructura de Directorios Principal

```
ui2code-rag/
  |-- data/                          # Datos del sistema
  |     |-- evaluate/                # Framework de evaluacion
  |     |     |-- docs_ui_code_en.jsonl
  |     |     |-- qrels_ui_code_en.csv
  |     |     `-- eval_retrieval_*.csv
  |     |-- generated_code/          # HTML generados
  |     |-- temp_images/             # Imagenes temporales
  |     `-- websight/                # 900 JSONs corpus
  |-- docs/                          # Documentacion y diagramas
  |     |-- diagrama-flujo-arquitectura-multi-agente.png
  |     `-- diagrama-flujo-rag-pipeline.png
  |-- logs/                          # Logs del sistema
  |-- src/
  |     |-- agents/
  |     |     |-- code_a2a_agent/    # Code Agent (p. 10001)
  |     |     |-- orchestator_agent/# Coordinador
  |     |     |-- rag_agent/         # RAG + WebSight
  |     |     `-- visual_a2a_agent/ # Visual Agent (p. 10000)
  |     |-- app/                     # Streamlit (p. 8501)
  |     |     |-- main.py
  |     |     |-- pages/             # Paginas multipagina
  |     |     `-- views/             # Logica de vistas
  |     `-- common/                  # Utilidades comunes
  |-- docker-compose.yaml
  |-- Dockerfile
  |-- Makefile
  |-- run.sh                  # Script de inicio automatico
  `-- requirements.txt
```

6. Flujo funcional - Interfaz de usuario

El sistema expone una aplicación web Streamlit multipágina accesible en `http://localhost:8501` con cinco secciones principales.

6.1. Página: Home

Funcionalidad: Dashboard principal con estado del sistema.
Características:

- Inicialización automática del corpus HTML/CSS (900 documentos WebSight)
- Botón de descarga del dataset si `data/websight/` está vacío
- Proceso: `ensure_rag_ready()` → `WebSightLoader.download_websight_data()`
- Navegación rápida a todas las funcionalidades
- Métricas del sistema: agentes activos, documentos cargados, índice Pinecone

6.2. Página: UI to Code (Principal)

Funcionalidad: Conversión de imágenes de diseños UI a código HTML/Tailwind CSS.

Flujo de uso:

1. **Upload de imagen**: Drag & drop o selección de archivo (PNG/JPG/JPEG/WebP)
2. **Instrucciones personalizadas** (opcional, bilingüe español/inglés):
   - Estilo: "Usar tema oscuro con acentos púrpura"
   - Responsive: "Hacer completamente responsive con diseño mobile-first"
   - Interactividad: "Agregar efectos hover sutiles y animaciones"
   - Accesibilidad: "Incluir etiquetas ARIA y alto contraste"
3. **Ejecución**: Click en "Generar código"
4. **Visualización de resultados**:
   - Análisis visual (componentes, layout, estilo, paleta)
   - Patrones HTML/CSS encontrados (expandibles con scores)
   - Código HTML generado (editor con syntax highlighting)
   - Preview HTML interactivo (iframe)
5. **Descarga**: Guardado automático en `data/generated_code/`

Procesamiento interno:

1. Orchestrator envía imagen a Visual Agent vía A2A
2. Visual Agent retorna análisis estructurado JSON
3. RAG Agent ejecuta búsqueda híbrida sobre corpus
4. Code Agent genera HTML usando análisis + patrones + instrucciones
5. Validación anti-hallucination con `_validate_html_components()`
6. Consolidación y presentación de resultados

6.3. Página: Query Interface

Funcionalidad: Búsqueda de patrones HTML/CSS usando RAG + generación desde prompt.

Modos de operación:

- **Modo búsqueda**: Recupera patrones similares del corpus
- **Modo Prompt→HTML**: Genera código desde descripción textual

Parámetros configurables:

- `top_k`: Número de resultados a mostrar (1-20)

- `top_retrieve`: Candidatos antes de re-ranking (5-200)
- `use_reranking`: Activar/desactivar Cross-Encoder
- `include_summary`: Resúmenes generados por IA

Visualización:

- Resultados clasificados con scores de relevancia
- Código fuente completo de cada patrón
- Metadata: tipo de documento, descripción, componentes
- Resumen contextual (si está habilitado)

6.4. Página: Evaluaciones

Funcionalidad: Framework de evaluación de rendimiento del sistema RAG.
Dataset de evaluación:

- `data/evaluate/docs_ui_code_en.jsonl`: 9 documentos HTML
- `data/evaluate/qrels_ui_code_en.csv`: Labels de relevancia

Configuración de evaluación:

- `k`: Valores para métricas (ej: 3, 5, 10)
- `top_retrieve`: Candidatos a recuperar (5-200)
- `top_final`: Resultados después de re-ranking (1-50)
- `device`: CPU/CUDA/MPS para procesamiento

Métricas calculadas:

- **Precision@k**: Proporción de resultados relevantes en top-k
- **Recall@k**: Proporción de documentos relevantes recuperados
- **nDCG**: Normalized Discounted Cumulative Gain (calidad de ranking)
- **MRR**: Mean Reciprocal Rank (posición del primer relevante)

Visualización:

- Tabla por query: métricas detalladas
- Tabla agregada: promedios macro por k
- Botones de descarga en CSV
- Comparación pre/post re-ranking

6.5. Página: System Status

Funcionalidad: Monitoreo en tiempo real del sistema.
Información mostrada:

- **Estado de agentes**:
  - Visual Agent (puerto 10000): Online/Offline
  - Code Agent (puerto 10001): Online/Offline
  - Verificación de endpoints A2A
- **Métricas del corpus**:
  - Total de documentos: 900 (WebSight)
  - Total de chunks indexados
  - Estado del índice Pinecone
- **Configuración activa**:
  - Modelos en uso (VISUAL_MODEL, CODE_MODEL)
  - Parámetros de chunking
  - Endpoints de agentes

6.6. Página: Corpus Information

Funcionalidad: Exploración del corpus HTML/CSS.
Características:

- **Navegación de documentos**:
  - Lista de 900 documentos WebSight
  - Filtrado por tipo (landing, dashboard, form, etc.)
- **Vista previa**:
  - Código HTML completo
  - Metadata: description, components, type
- **Estadísticas**:
  - Distribución de tipos de documento
  - Longitud promedio de documentos
  - Estadísticas de fragmentación (chunks)

7. Gobernanza de calidad y anti-alucinación

7.1. Estrategias implementadas

7.1.1. RAG como contexto obligatorio

Los prompts del Code Agent incluyen citas explícitas de patrones recuperados para "anclar" la generación:

- Patrones formateados con primeros 2000 caracteres de HTML
- Referencias directas a estructura técnica de ejemplos
- Instrucciones para usar patrones como guía, no como copia literal

7.1.2. Guardrails de validación

Sistema de validaciones en múltiples capas:

Cuadro 2: Guardrails Implementados

Guardrail	Hub	Función
`valid_json`	Sí	Verifica formato JSON válido (no schema)
`web_sanitization`	Sí	Detecta y escapa caracteres sospechosos (librería `bleach`)
`valid_schema_json`	No	Valida schema Pydantic específico (custom del equipo)
`is_html_field`	No	Valida HTML con BeautifulSoup (custom del equipo)

7.1.3. Validación anti-hallucination en Code Agent

Método `_validate_html_components()` implementado en Code Agent:

- **Detección**: Identifica secciones HTML extra (header, nav, footer, aside)
- **Comparación**: Contrasta con componentes del análisis visual
- **Logging**: Warnings detallados cuando hay discrepancias
- **Ejemplo**: Si el análisis visual pide "login form", pero el código genera "landing page completa con header y footer", el sistema logea una advertencia

7.1.4. Prompts optimizados con reglas críticas

Extracto del prompt del Code Agent (`src/agents/code_agent/src/texts/prompts.py`):

Listing 2: Fragmento de Prompt Anti-Hallucination

```
CRITICAL RULES:
1. Generate ONLY components mentioned in visual analysis
2. DO NOT add header, nav, footer unless explicitly requested
3. Use patterns as technical reference, not literal base
4. If visual analysis says "login␣form", generate ONLY login
   form
5. DO NOT hallucinate complete landing pages from single
   components
```

7.1.5. Fallback controlado

Cuando la evidencia visual es insuficiente:

- Sistema detecta baja confianza en análisis (imagen borrosa, ambigua)
- En lugar de alucinar, comunica: *"Evidencia insuficiente para generar código. Por favor, proporcione una imagen más nítida y especifique 2-3 componentes clave."*
- Evita generación de código no utilizable

7.1.6. Umbrales de confianza en recuperación

- Si el recall efectivo del RAG cae bajo umbral configurado
- Sistema solicita: mayor nitidez de imagen O instrucciones adicionales
- Previene generación con contexto pobre

7.2. Hallazgos clave

1. **Few-shots + patrones RAG reducen alucinación**: Restringir la libertad del LLM con ejemplos concretos y patrones recuperados disminuye significativamente la alucinación de layouts complejos.
2. **Esquemas estrictos evitan salidas mal formadas**: Los schemas Pydantic/-Guardrails previenen respuestas fuera de contrato, pero pueden introducir errores de validación si el LLM se desvía (ej: listas vacías, campos omitidos).
3. **Nitidez de imagen es crítica**: La calidad visual de la imagen de entrada impacta directamente en la fidelidad del código generado.
4. **Re-ranking estabiliza prompts largos**: El Cross-Encoder mejora la estabilidad del top-k y reduce el drift en prompts con mucho contexto.
5. **Separación de agentes facilita debugging**: La arquitectura multi-agente permite prompts más simples por agente y debugging localizado.

8. Implementación y tooling

8.1. Tecnologías core

Cuadro 3: Stack tecnológico

Componente	Tecnología	Versión
Modelos IA	OpenRouter	API v1
Embeddings	Sentence-Transformers	$\geq$ 5.1.1
Vector DB	Pinecone	$\geq$ 3.0.0
Web UI	Streamlit	$\geq$ 1.28.0
A2A Protocol	a2a-sdk	$\geq$ 0.3.8
Guardrails	guardrails-ai	$\geq$ 0.6.7
Logging	loguru	$\geq$ 0.7.3
ASGI Server	Uvicorn	$\geq$ 0.37.0

8.2. OpenRouter integration

Ventajas:

- Acceso unificado a múltiples proveedores (Google, Anthropic, DeepSeek, OpenAI)
- Modelos gratuitos disponibles (Gemini Flash, DeepSeek R1 Distill)
- Modelos premium bajo demanda
- Gestión automática de rate limits y fallbacks

Modelos recomendados:

Listing 3: Configuracion de Modelos en .env

```
# Modelos gratuitos (recomendados)
VISUAL_MODEL=google/gemini-2.0-flash-exp:free
CODE_MODEL=deepseek/deepseek-r1-distill-llama-70b:free

# Alternativas premium (mejor calidad)
# VISUAL_MODEL=anthropic/claude-3.5-sonnet
# CODE_MODEL=deepseek/deepseek-r1
```

8.3. Pinecone vector database

Configuración:

- Index: `pln3-index`
- Namespace: `html-css-examples` (corpus), `eval-metrics` (evaluación)
- Dimensión: 384 (all-MiniLM-L6-v2)
- Métrica: Cosine similarity

Proceso de indexación:

1. WebSightLoader carga 900 documentos
2. Chunking con max_tokens=400, overlap=100
3. Generación de embeddings con Sentence-Transformers
4. Upload automático a Pinecone (solo si no existe)
5. Verificación de conectividad en System Status

8.4. Streamlit como UI acelerada

Características utilizadas:

- Arquitectura multipágina con `st.Page`
- Componentes interactivos: `st.file_uploader`, `st.text_area`, `st.expander`
- Columnas responsivas con `st.columns`
- Session state para persistencia
- Tema custom (paleta violeta) en `.streamlit/config.toml`

8.5. Uvicorn + Starlette + A2A

Agentes A2A:

- Visual Agent y Code Agent exponen endpoints JSONRPC
- Servidor ASGI con Uvicorn
- Comunicación asíncrona (aunque actualmente síncrona en ejecución)
- Agent cards en `/.well-known/agent-card.json`

Ejemplo de comunicación A2A:

Listing 4: Llamada A2A desde Orchestrator

```
# Orchestrator envia mensaje a Visual Agent
response = await orchestrator.send_message_to_visual_agent(
    image_path="data/temp_images/upload_12345.jpg"
)

# Visual Agent procesa y retorna JSON
visual_analysis = response["result"]["analysis"]
# {"components": [...], "layout": "...", "style": "...", ...}
```

8.6. Configuración dinámica

pyprojroot + YAML:

- `src/config.py`: Resuelve rutas relativas portables
- `src/config.yaml`: URLs de agentes, timeouts, parámetros
- Variables de entorno con python-dotenv

Ejemplo de configuración:

Listing 5: config.yaml - Endpoints Agentes

```
agents:
  visual_agent:
    url: http://localhost:10000
    timeout: 60
  code_agent:
    url: http://localhost:10001
    timeout: 120
```

8.7. Scripts de evaluación

Ubicación: `src/agents/rag_agent/rag/evaluators/evaluate_retrieval.py`
Uso:

Listing 6: Ejecucion de Evaluacion

```
python -m src.agents.rag_agent.rag.evaluators.
   evaluate_retrieval \
  --docs data/evaluate/docs_ui_code_en.jsonl \
  --qrels data/evaluate/qrels_ui_code_en.csv \
  --ks 3,5 \
  --top_retrieve 10 \
  --top_final 5
```

Salida:

- `eval_retrieval_per_query.csv`: Métricas detalladas
- `eval_retrieval_aggregated.csv`: Promedios macro

9. Organización del proyecto y división del trabajo

9.1. Distribución de responsabilidades

Cuadro 4: Matriz de Contribuciones del Equipo

Componente	**Jorge**	**Bruno**	**Fabricio**	**Noelia**
Estructura general del proyecto	D	D	C	C
Dockerización de la solución	C	D	C	D
Modelo de visión por computadora	D	D	D	D
Modelo de generación de código	D	C	C	C
Sistema de logging (loguru)	C	C	D	C
Arquitectura A2A	C	D	C	C
Guardrails	C	C	D	D
Corpus y generación de RAG	D	C	D	C
Interfaz visual (Streamlit)	C	C	C	D
TOTAL	D:4, C:5	D:4, C:5	D:4, C:5	D:4, C:5
Horas insumidas	40 hs	40 hs	40 hs	40 hs

Leyenda: D = Desarrollo (responsabilidad principal), C = Colaboración (soporte y revisión)

9.2. Metodología de trabajo

- **Gestión**: Sprints semanales con reuniones de sincronización
- **Control de versiones**: Git con feature branches
- **Revisión de código**: Pull requests con revisión cruzada
- **Comunicación**: Slack + reuniones virtuales
- **Documentación**: Markdown en repositorio + comentarios en código

10. Resultados y estado actual

10.1. Casos positivos

- **UIs con estructura clara**: El sistema genera HTML/Tailwind coherente con grid/flex correctos, spacings razonables y componentes bien definidos (cards, buttons, forms).
- **Fallback seguro**: Ante imágenes borrosas o ambiguas, el sistema comunica proactivamente: *"Evidencia insuficiente para generar código. Proporcione mejor nitidez y especifique 2-3 componentes clave."*
- **RAG aporta consistencia**: Los snippets recuperados mejoran sustancialmente la consistencia estilística del código generado.

- **Código artesanal**: El HTML generado no depende de icon libraries externas, usando elementos geométricos simples y tipografía creativa.
- **Personalización bilingüe**: Las instrucciones en español o inglés funcionan correctamente para customizar el output.

10.2. Limitaciones observadas

1. **Estados de UI no reseteados**: En ciertos flujos, botones o controles quedaban deshabilitados por estados no correctamente reinicializados en Streamlit.
2. **Validación de schemas**: El Code Agent puede fallar si el modelo genera listas vacías u omite campos esperados (ej: `components=[]`), cortando la interacción.
3. **Excepciones en orquestación A2A**: Se observaron excepciones de `nest-asyncio` cuando respuestas parciales no respetan el contrato JSONRPC.
4. **Layout en Query Interface**: El output de búsqueda aparecía en columna lateral en lugar de debajo del textarea (ajuste de layout en Streamlit).
5. **Modelos gratuitos con limitaciones**: Los modelos free de OpenRouter tienen rate limits que pueden causar delays en desarrollo intensivo.

10.3. Resultados de evaluación del sistema RAG

Cuadro 5: Métricas de evaluación para diferentes valores de k

k	P@k Pre	R@k Pre	nDCG Pre	P@k Post	R@k Post	nDCG Post	MRR Pre	MRR Post
3	0.30	0.90	0.79	0.30	0.90	0.86	0.77	0.87
5	0.20	1.00	0.83	0.20	1.00	0.90	0.77	0.87

10.3.1. Significado de las métricas

- **k**: Número de resultados considerados (top-k)
- **Precision@k (Pre/Post)**: Proporción de resultados relevantes en los primeros k (antes/después de re-ranking)
- **Recall@k (Pre/Post)**: Proporción de documentos relevantes recuperados en los primeros k
- **nDCG (Pre/Post)**: Normalized Discounted Cumulative Gain - calidad del ranking (mayor es mejor)
- **MRR (Pre/Post)**: Mean Reciprocal Rank - posición del primer resultado relevante (mayor es mejor)

10.3.2. Interpretación de resultados

1. **Alto recall (0.9 - 1.0)**: El sistema recupera casi todos los documentos relevantes en el top-k, indicando buena cobertura.
2. **Precisión moderada (0.2 - 0.3)**: Hay presencia de resultados no relevantes en el top-k, sugiriendo espacio de mejora en filtrado.
3. **Mejora por re-ranking**: El nDCG y MRR aumentan después del re-ranking (0.79→0.86 y 0.77→0.87), demostrando que el Cross-Encoder posiciona mejor los relevantes.
4. **Recall no afectado por re-ranking**: Como esperado, el re-ranking no cambia qué documentos se recuperan, solo su orden.
5. **Estabilidad en k=5**: Con k=5 se logra recall perfecto (1.0) manteniendo precisión razonable.

10.3.3. Resumen de evaluación

- El sistema recupera casi todos los documentos relevantes (alto recall)
- La precisión podría mejorar con refinamiento de embeddings o ajuste de pesos BM25/Vector
- El re-ranking con Cross-Encoder mejora significativamente la posición de los relevantes
- Los resultados validan la efectividad del enfoque híbrido BM25 + Vector + Re-ranking

11. Lecciones aprendidas

1. **RAG + guardrails > LLM puro**: Para la tarea de UI-to-Code, un sistema RAG con validaciones es significativamente superior a un LLM sin contexto.
2. **Contratos de salida reducen alucinación**: Los schemas Pydantic/Guardrails fuerzan estructura, pero requieren diseño cuidadoso con campos opcionales y defaults para evitar rupturas de flujo.
3. **Calidad de input es crítica**: Pedir al usuario 2-3 componentes clave y una imagen nítida mejora sustancialmente la fidelidad del código generado.
4. **Re-ranking estabiliza prompts largos**: El Cross-Encoder reduce drift en prompts con mucho contexto, mejorando consistencia.
5. **Separación de agentes facilita debugging**: La arquitectura multi-agente permite prompts más simples por componente y debugging localizado, aunque introduce complejidad de orquestación.
6. **Docker simplifica despliegue**: La contenerización resolvió problemas de reproducibilidad entre entornos de desarrollo.

7. **Modelos gratuitos son viables**: OpenRouter con modelos free (Gemini Flash, DeepSeek) permite MVP funcional, aunque con limitaciones de rate limits.

8. **Logging estructurado es esencial**: Loguru facilitó debugging de flujos complejos multi-agente.

12. Roadmap propuesto

12.1. Corto plazo (1-3 meses)

12.1.1. Mitigación de alucinación y UX

- Prompts con "chain-of-thought estructurada" no-verbatim para que Visual Agent reporte evidencias explícitas (colores, jerarquías, distancias relativas)
- Refuerzo con few-shots curados de HTML/Tailwind y contra-ejemplos (qué NO hacer)
- Validación incremental: primero estructura mínima (layout), luego detalles (estilos), evitando all-or-nothing del schema final
- Mejoras de UI: mover resultados siempre debajo del textarea, spinners por paso, mensajes de error accionables

12.1.2. RAG y corpus

- Aumentar corpus HTML/CSS a 2000+ documentos con ejemplos etiquetados por componentes y layouts (dashboard, e-commerce, forms)
- Embeddings específicos para código (e.g., code-aware models como CodeBERT)
- Re-ranking por coverage de componentes solicitados
- Persistir metadatos (tokens, longitud, nivel de nesting) para re-ranking semántico más fino

12.2. Mediano plazo (3-6 meses)

12.2.1. Agentes y orquestación

- Retries con backoff exponencial y degradación graciosa cuando LLM viola contrato
- Relleno automático de campos faltantes en respuestas parciales
- Mensajería asincrónica real con tasks para cada generación (aprovechar A2A async)
- Implementación de MCP (Model Context Protocol) con tooling para RAG Agent
- LLM como orquestador dinámico en lugar de orquestador determinista
- Mejorar capa de persistencia (actualmente todo en memoria)
- Tests de contratos (Pydantic) per-commit con golden outputs

12.2.2. Evaluación y calidad

- Definir conjunto de imágenes canónicas con métricas de fidelidad:
  - Component coverage score
  - Layout IoU simplificado
  - Developer acceptance testing
- Automatizar batería de regresión con capturas y hashes de HTML resultante
- Tests robustos (actualmente solo MVP sin coverage completo)

12.2.3. Costos y observabilidad

- Estimación y tracking de costos por invocación
- Conteo de tokens utilizados con alertas de uso
- Telemetría con OpenTelemetry para observabilidad distribuida
- Dashboard de métricas de negocio (tiempo de generación, tasa de éxito, etc.)

12.3. Largo plazo (6+ meses)

- Fine-tuning de modelos propios para Visual y Code Agents
- Integración con IDEs (VSCode extension)
- Soporte para frameworks adicionales (React, Vue, Angular)
- Sistema de feedback loop para mejora continua del corpus
- Multi-tenancy para empresas con estilos diferentes

13. Desafíos afrontados durante el trabajo

13.1. Elección de modelos

Problema: Por restricción presupuestaria, los miembros del equipo no contaban con acceso a modelos pagos de forma continua.

Solución:

- Optar por modelos gratuitos de OpenRouter (Gemini Flash, DeepSeek R1 Distill)
- Enfrentar volatilidad: OpenRouter daba de baja modelos con frecuencia
- Estrategia de equipo: rotar entre diferentes modelos gratuitos, todos con resultados aceptables
- Implementar configuración flexible vía .env para cambiar modelos sin modificar código

13.2. Ejecución en pipeline de procesos

Problema: Al comienzo se usaba Makefile para levantar agentes, servers y artefactos del sistema. Esto era tedioso y requería múltiples terminales, haciendo lenta la etapa de desarrollo y QA.

Solución:

- Dockerizar la solución completa con Docker Compose
- Crear script `run.sh` para automatizar configuración y despliegue
- Resultado: funcionamiento rápido y eficaz en todas las máquinas de desarrollo
- Beneficio adicional: portabilidad para despliegue en producción

13.3. Gestión de configuraciones

Problema: Múltiples archivos .env (raíz, Visual Agent, Code Agent) causaban confusión sobre qué variables usar dónde.

Solución:

- Script `run.sh` guía interactivamente la configuración
- Documentación clara en README sobre cada archivo .env
- Validación de configuraciones al inicio de cada agente

13.4. Sincronización de versiones de librerías

Problema: Conflictos de dependencias entre Streamlit, Guardrails y Sentence-Transformers.

Solución:

- Usar `pyproject.toml` con Poetry/UV para resolución de dependencias
- Bloquear versiones críticas con constraints
- Separar dependencias de desarrollo y producción

13.5. Performance de evaluaciones

Problema: Evaluaciones con 900 documentos y re-ranking eran lentas en CPUs locales.

Solución:

- Parametrizar device (CPU/CUDA/MPS) en evaluaciones
- Cachear resultados de búsqueda vectorial
- Subconjunto de evaluación (9 docs) para pruebas rápidas

14. Conclusiones

El enfoque multi-agente combinado con RAG híbrido y guardrails ha demostrado ser efectivo para reducir alucinación y mejorar la utilidad del código HTML/Tailwind generado a partir de imágenes de interfaz de usuario.

14.1. Logros principales

- **Sistema funcional end-to-end**: UI operativa con flujo completo desde imagen hasta código descargable
- **RAG híbrido probado**: BM25 + Vector Search + Re-ranking con métricas validadas (Recall 0.9-1.0, nDCG mejora 7-9 %)
- **Arquitectura A2A robusta**: Comunicación JSONRPC entre agentes con fallbacks
- **Validaciones efectivas**: Guardrails custom y del Hub reducen alucinaciones
- **Tooling completo**: Evaluación, logging, monitoring, UI multipágina
- **Despliegue simplificado**: Docker Compose + script automático

14.2. Desafíos persistentes

- Validación robusta de schemas con tolerancia a errores
- Estado de UI y sincronización en Streamlit
- Operatoria asincrónica real entre agentes (actualmente síncrono)
- Limitaciones de modelos gratuitos (rate limits, disponibilidad)

14.3. Valor del proyecto

El sistema ya ofrece una base funcional y extensible para:

- Acelerar desarrollo frontend con código estandarizado
- Reducir deuda técnica mediante reutilización de patrones
- Servir como referencia de arquitectura multi-agente con RAG
- Facilitar investigación en mitigación de alucinaciones

Con el roadmap propuesto, el sistema puede evolucionar hacia mayor fidelidad visual, estabilidad operativa y medición reproducible de calidad, acercándose a un producto comercialmente viable.

14.4. Próximos pasos inmediatos

1. Expandir corpus a 2000+ documentos
2. Implementar validación incremental (layout → estilos)
3. Añadir telemetría de costos y tokens
4. Tests de regresión automatizados
5. Documentar casos de uso con empresas piloto

15. Anexo - Setup y operación

15.1. Requisitos previos

15.1.1. Producción (recomendado)

- Docker y Docker Compose instalados
- Puertos disponibles: 8501, 10000, 10001
- APIs keys:
  - OpenRouter API Key (https://openrouter.ai/settings/keys)
  - Pinecone API Key (https://pinecone.io/)
  - Guardrails API Key (https://hub.guardrailsai.com/keys)

15.1.2. Desarrollo

- Python 3.10 - 3.12
- pip o UV para gestión de paquetes
- Entorno virtual (venv o conda)

15.2. Instalación y ejecución - Producción

Paso 1: Ejecutar script de configuración automática

```
bash run.sh
```

Paso 2: El script realizará:

1. Verificación de Docker y Docker Compose
2. Solicitud interactiva de API keys
3. Creación de archivos .env en ubicaciones correctas:
   - `.env` (raíz): PINECONE_API_KEY
   - `src/agents/visual_a2a_agent/.env`: OPENROUTER_API_KEY, VISUAL_MODEL, GUARDRAILS_API_KEY

- `src/agents/code_a2a_agent/.env`: OPENROUTER_API_KEY, CODE_MODEL, GUARDRAILS_API_KEY

4. Ejecución de `docker compose up --build`

Paso 3: Acceder a la aplicación

- URL: http://localhost:8501
- Página principal: UI to Code

Nota: La primera ejecución puede tardar ∼30 minutos descargando imágenes Docker (∼30 GB).

15.3. Instalación y ejecución - Desarrollo

Paso 1: Instalar dependencias

```
# Crear entorno virtual
python -m venv venv
source venv/bin/activate  # Windows: venv\Scripts\activate

# Opcion 1: UV (mas rapido)
uv sync

# Opcion 2: Poetry
poetry install

# Opcion 3: pip
pip install -e .
```

Paso 2: Configurar variables de entorno

```
# Copiar plantillas
cp .env.example .env
cp src/agents/visual_a2a_agent/.env.example \
   src/agents/visual_a2a_agent/.env
cp src/agents/code_a2a_agent/.env.example \
   src/agents/code_a2a_agent/.env

# Editar cada archivo .env con tus API keys
```

Paso 3: Configurar Guardrails

```
# Configurar Guardrails
make run-guardrails-configuration

# O manualmente
guardrails configure
guardrails hub install hub://guardrails/regex_match
guardrails hub install hub://guardrails/valid_json
guardrails hub install hub://guardrails/web_sanitization
```

Paso 4: Iniciar agentes (3 terminales separadas)

```
# Terminal 1: Visual Agent
make run-visual-agent

# Terminal 2: Code Agent
make run-code-agent

# Terminal 3: Streamlit
make run-server
```

Paso 5: Acceder a http://localhost:8501

15.4. Comandos make disponibles

```
# Descargar dataset WebSight
make download-websight

# Configurar Guardrails
make run-guardrails-configuration

# Iniciar agentes
make run-visual-agent     # Puerto 10000
make run-code-agent       # Puerto 10001
make run-server           # Puerto 8501

# Evaluacion de retrieval
make evaluate-retrieval
```

15.5. Guardrails - Descripción técnica

Guardrails UI es una interfaz para supervisar, gestionar y controlar aplicaciones impulsadas por IA, especialmente modelos de lenguaje. Proporciona un entorno visual donde los desarrolladores pueden definir reglas, límites y flujos de trabajo, asegurando que las respuestas cumplan con criterios de seguridad, precisión y consistencia.

Guardrail Hub es una plataforma centralizada para gestionar, compartir y reutilizar conjuntos de guardrails, facilitando el acceso a plantillas predefinidas y manteniendo consistencia en múltiples proyectos.

15.5.1. Guardrails utilizados en el proyecto

Cuadro 6: Guardrails implementados

Nombre	Hub	Agentes	Descripción
ValidJson	Sí	code, visual	Verifica formato JSON válido. No valida schema específico.
WebSanitization	Sí	code	Detecta y escapa caracteres sospechosos. Basado en la librería `bleach`.
ValidSchemaJson	No	code, visual	Valida schema Pydantic predeterminado. Extensión de ValidJson creada por el equipo.
IsHTMLField	No	code	Verifica formato HTML válido. Basado en BeautifulSoup. Creado por el equipo.

15.6. Sistema de logging

Librería: Loguru
Características:

- Logs estructurados con colores
- Rotación automática por fecha
- Niveles configurables (DEBUG, INFO, WARNING, ERROR)
- Salida a consola y archivo

Ubicación de logs:

- Desarrollo: consola del terminal
- Producción: `logs/ui2code.YYYY-MM-DD.log`
- Docker: logs del contenedor (accesibles con `docker logs`)

Información logueada:

- Respuestas completas de modelos
- Errores de validación de guardrails
- Warnings de anti-hallucination
- Tiempos de ejecución de cada agente
- Métricas de búsqueda RAG (scores, latencias)

15.7. Dataset WebSight - Detalles técnicos

Fuente: HuggingFace (`HuggingFaceM4/WebSight`)

Cuadro 7: Características del Dataset WebSight

Campo	Descripción
Nombre completo	WebSight (parte del proyecto HuggingFaceM4)
Modalidades	Imágenes, Texto
Formato	Parquet (convertido a JSON localmente)
Idioma	Inglés
Tamaño total	Versión v0.2: 1.92 millones de filas; v0.1: 823 mil filas
Split	Solo "train" (entrenamiento)
Longitud de texto	`text`: 344-6,310 caracteres; `llm_generated_idea`: 25-566 caracteres
Licencia	Creative Commons BY 4.0 (cc-by-4.0)

Campos utilizados:

- **text**: Contiene código HTML completo. Se usa para:
  - Crear archivos .html en `data/websight/`
  - Generar chunks (max_tokens=400, overlap=100)
  - Crear embeddings y subirlos a Pinecone
- **llm_generated_idea**: Descripción textual del componente. Se usa en:
  - Modo evaluación para verificar recuperación relevante
  - Metadata de documentos para contexto semántico

Subconjunto utilizado:
Si bien el dataset contiene casi 2 millones de registros, se decidió utilizar un subconjunto de **900 documentos** por:

- Restricciones de la capa gratuita de Pinecone
- Evitar tiempos excesivos de descarga (dataset completo > 50GB)
- Fines académicos del proyecto
- Balance entre cobertura y rendimiento

Expansión futura: Se espera ampliar a 2000+ documentos en versión comercial para mejorar cobertura de patrones.

15.8. Troubleshooting

15.8.1. Si no ves 900 documentos en el corpus

1. Verifica que `data/websight/` contenga archivos JSON
2. Si está vacío, ejecuta desde Home de Streamlit: "Descargar dataset WebSight"
3. O manualmente: `python src/scripts/download_websight.py`
4. Reinicia Streamlit - el corpus se carga automáticamente

15.8.2. Si los agentes no inician

1. Verifica archivos .env:
   - `src/agents/visual_a2a_agent/.env` debe tener `OPENROUTER_VISUAL_MODEL`
   - `src/agents/code_a2a_agent/.env` debe tener `OPENROUTER_CODE_MODEL`
   - `.env` raíz debe tener `PINECONE_API_KEY`
2. Usa `bash run.sh` para configuración automática
3. Verifica puertos 10000, 10001, 8501 disponibles

15.8.3. Si hay alucinaciones en código generado

- Revisa logs del Code Agent - debe mostrar warnings de validación
- Verifica que el análisis visual sea preciso
- Usa imagen más nítida y proporciona 2-3 componentes clave
- Considera modelos más potentes (DeepSeek R1 70B, Claude 3.5)

15.8.4. Si Pinecone no funciona

1. Verifica `PINECONE_API_KEY` en `.env` raíz
2. Confirma que el índice existe y acepta dimensión 384
3. Namespaces usados: `html-css-examples` (corpus), `eval-metrics` (evaluación)
4. Verifica conectividad en página "System Status"

16. Referencias

1. HuggingFace M4. (2024). *WebSight Dataset*. https://huggingface.co/datasets/HuggingFaceM4/WebSight
2. Guardrails AI. (2024). *Guardrails Hub Documentation*. https://hub.guardrailsai.com/

3. Pinecone. (2024). *Pinecone Vector Database Documentation*. https://docs.pinecone.io/

4. OpenRouter. (2024). *OpenRouter API Documentation*. https://openrouter.ai/docs

5. Streamlit. (2024). *Streamlit Documentation*. https://docs.streamlit.io/

6. A2A Protocol. (2024). *Agent-to-Agent Communication Protocol*. https://github.com/a2a-protocol/a2a

7. Robertson, S., & Zaragoza, H. (2009). *The Probabilistic Relevance Framework: BM25 and Beyond*. Foundations and Trends in Information Retrieval, 3(4), 333-389.

8. Reimers, N., & Gurevych, I. (2019). *Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks*. EMNLP 2019.