Resumen rápido

Construir un **sistema de búsqueda distribuido** que:

- Crawlee 500+ páginas y extraiga título, descripción, keywords y enlace.
- Publique items en una cola (Kafka).
- Almacene metadatos en **MySQL**, contenido en **MongoDB** e índices en **PostgreSQL**.
- Permita consultas distribuidas que consulten los 3 nodos y combinen resultados.
- Interfaz web con **Next.js** para búsquedas y listados.
- Todo en **monorepo**, contenerizado con **Docker** / Docker Compose (simular nodos distribuidos).
- Tecnologías de scraping: **Scrapy** + **Requests/BeautifulSoup** (Python). Backend de orquestación / API: **FastAPI (Python)** o **Node.js** para el gateway; usar **Node.js** donde se pide explícitamente (Next.js y posible gateway).

Principios para entregarlo rápido

1. **MVP mínimo y comprobable**: Priorizar flujo completo (crawler → Kafka → almacenamiento → consulta → UI) sobre optimizaciones (ranking, stemming, búsqueda avanzada).
2. **Paralelizar trabajo**: equipos/roles paralelos: Crawler (Python), Ingest/Storage (Python), API Gateway / Agregador (Node.js), Frontend (Next.js), Infra (Docker/Kafka/DB).
3. **Simular distribución con containers**: instancias replicadas en Docker Compose (p. ej. mysql1/mysql2, postgres1/postgres2, mongodb1/mongodb2) para demostrar tolerancia a fallos.
4. **Reglas de corte**: si una funcionalidad consume tiempo (p. ej. ranking TF-IDF), dejarlo como mejora; entregar una búsqueda por coincidencia simple y orden por fecha/TF básico.

Roadmap (día a día hasta la entrega — fecha del enunciado: 10 de noviembre)

> Hoy es 7 de noviembre, por lo que propondré **3 sprints (Día 0, Día 1, Día 2)** — enfoque máximo pragmático.

Día 0 — Preparación y scaffolding (4–6 horas)

- Crear **monorepo** y estructura base (detallo estructura más abajo).
- Configurar **Docker Compose** con servicios mínimos (zookeeper/kafka, mysql, postgres, mongo, kafka-connect opcional, kafka-ui opcional).
- Crear repositorio, issues y board (GitHub Projects/Jira).
- Skeletons:
  - `crawler/` (Scrapy project).
  - `ingest/` (Python service que lee Kafka y escribe en DBs).

  - `api/` (Node.js gateway + endpoints de búsqueda).
  - `web/` (Next.js app).
  - `infra/` (docker-compose.yml, scripts de bootstrap).

- Validar conexión a Kafka desde un contenedor Python y Node.

Día 1 — Pipeline básico de extremo a extremo (8–12 horas)

- **Crawler (Python, Scrapy)**:
  - Spider que recorre lista de URLs (seed list) y extrae título, meta-description, keywords y body text.
  - Envío de mensajes a **Kafka** (topic `pages.raw`).
- **Ingest (Python)**:
  - Consumidor Kafka que procesa mensajes: limpia texto, extrae campos, genera `document_id`.
  - Inserta:
    - Metadatos en **MySQL** (`documents` tabla: id, url, title, description, crawl_ts, source`).
    - Contenido completo en **MongoDB** (`documents` collection: _id, document_id, content, raw_html`).
    - Índice / tokens básicos en **PostgreSQL** (`inverted_index` simple o tabla con token, doc_id, freq). (Si falta tiempo, almacenar tokens en Postgres como JSONB).
- **API / Agregador (Node.js)**:
  - Endpoint `/search?q=...` que:
    1. Envía la query a cada nodo (via DB queries: MySQL metadata + Postgres indices + Mongo for content snippets) — o mejor: aggregator consulta Postgres (index), luego recupera metadata from MySQL and snippet from Mongo.
    2. Combina y normaliza resultados.
    3. Soporta fallback si un nodo está caído (mostrar resultados parciales y marcar origen).
- **Web (Next.js)**:
  - Página de búsqueda simple: caja de búsqueda, listado con título, link y snippet.
- Pruebas básicas: crawl 500 páginas (o simular con seeds + sitemap), verificar que aparecen en UI.

Día 2 — Robustez, replicación y presentación (8–12 horas)

- **Replicación/Partición**:
  - Configurar réplicas mínimas (por ejemplo, mysql primary + replica, mongo replica set con 1 primario + 1 secundario, postgres con dos instancias o usar tablas particionadas). Simulación con Docker Compose.
  - Documentar estrategia de fragmentación: por dominio (hash(url) % N) y replicación: 2 réplicas por shard.

- **Tolerancia a fallos**:
    - Pruebas de caída: apagar contenedor Mongo o MySQL y validar que API sigue respondiendo con resultados parciales.
- **Medición de eficiencia**:
    - Métricas simples: tiempo medio de consulta, throughput ingest (docs/s), latencias.
    - Script de carga (k6 o simple Python) para evaluar consultas distribuidas.
- **Polishing**:
    - Autenticación básica (user/pass en API).
    - Documentación: arquitectura, diagrama topológico, ER básico, pasos de despliegue.
    - Preparación de presentación / demo.

Entregables (qué presentar)

1. Repositorio monorepo con código y Docker Compose funcional.
2. Docker Compose que arranque: Kafka, MySQL, Postgres, Mongo, crawler, ingest, api, web.
3. Demo funcional: crawl → index → búsqueda en UI.
4. Documentación: arquitectura, diagrama ER / topología, decisiones de fragmentación/replicación, pruebas de tolerancia a fallos y resultados (latencias/throughput).
5. Slides para defensa y script de demo (tips: mostrar caída de 1 nodo y resultados parciales).

Estructura de monorepo (sugerencia práctica)

```
/monorepo
├─ infra/
│  ├─ docker-compose.yml
│  ├─ kafka/
│  ├─ scripts/ (bootstrap sql, create topics)
├─ services/
│  ├─ crawler/         # Scrapy project (Python)
│  ├─ ingest/          # consumer -> writes to MySQL/Mongo/Postgres
(Python, FastAPI optional)
│  ├─ api/             # Node.js or FastAPI aggregator (Node.js preferred
for Next.js compat)
│  ├─ web/             # nextjs app (UI)
│  ├─ utils/           # shared utils, schemas, proto (if needed)
├─ docs/
│  ├─ architecture.md
│  ├─ er-diagrams/
├─ tests/
│  ├─ load_tests/
├─ .env.example
├─ README.md
```

Notas:

- Mantener cada servicio con Dockerfile propio.
- Usar `make` o scripts `infra/start.sh` para levantar todo y `infra/bootstrap.sh` para crear topics y esquemas.

Diseño de componentes clave (alto nivel)

- **Crawler (Scrapy)**:
  - Pipeline: Parse → Extract fields → Send to Kafka topic `pages.raw`.
  - Exporter Kafka: usar `confluent_kafka` o `kafka-python`.
- **Kafka**:
  - Topics: `pages.raw`, `pages.processed` (opcional).
  - Partitioning por hash(url) para paralelismo.
- **Ingest**:
  - Consumer: consume `pages.raw`, normaliza, tokeniza (token simple), guarda en MySQL/Mongo/Postgres.
- **Almacenamiento**:
  - MySQL: tablas relacionales para metadatos y administración.
  - MongoDB: contenido crudo y snippets.
  - PostgreSQL: índices invertidos / tokens (usar jsonb o tablas token→doc→freq).
- **Agregador**:
  - Node.js service que orquesta las consultas paralelas (promises), combina, ordena por score simple (match count + recency).
- **UI (Next.js)**:
  - SSR o client-side fetch hacia API `/api/search`, mostrar resultados paginados.

Estrategia de fragmentación y replicación (resumen)

- **Fragmentación**: hash por dominio/url → shard N (permite queries parciales por shard).
- **Replicación**: 2 réplicas por shard (primario/secundario). En Docker se simula con múltiples containers.
- **Consistencia**: enfoque eventual — escribir primero en Mongo + MySQL, luego actualizar índices en Postgres; documentar ventana de inconsistencia aceptada.

- **Failover**: API detecta nodos caídos y consulta réplicas; si índice no disponible, extrae por metadata+snippet.

Métricas y evaluación (qué medir para la entrega)

- Volumen: páginas indexadas (objetivo ≥ 500).
- Latencia de ingest: ms/doc o docs/s.
- Latencia de consulta: p50/p95.
- Robustez: prueba de caída de 1 nodo — sistema sigue entregando resultados.
- Correctitud: porcentaje de resultados con snippet y link válidos.

Lista mínima de tareas (checklist rápido)

- [ ] Crear monorepo + docker-compose base.
- [ ] Configurar Kafka y topics.
- [ ] Implementar Scrapy spider + envío a Kafka.
- [ ] Implementar ingest consumer (MySQL, Mongo, Postgres).
- [ ] Implementar esquema básico en MySQL/Postgres.
- [ ] Implementar Node.js API `/search`.
- [ ] Implementar Next.js frontend.
- [ ] Pruebas: crawl 500, búsqueda, falla de un nodo.
- [ ] Documentación + slides.

Riesgos y mitigaciones

- **Tiempo**: riesgo alto. Mitigación: cortar features no esenciales (ranking avanzado, paginación compleja).
- **Conexiones entre contenedores**: usar healthchecks y logs; preparar scripts para recrear topics.
- **Scraping bloqueado**: usar `robots.txt` y reducir velocidad; si sitios bloquean, ampliar seed list o usar mirrors.

Tips rápidos para acelerar el desarrollo

- Reusar bibliotecas: `kafka-python`, `pymysql`, `motor` (async mongo), `psycopg2` o `asyncpg`.
- Para el índice en Postgres, usa `JSONB` si no tienes tiempo para esquema invertido pesado.
- Para la UI, hacer SSR mínimo con fetch a tu API; no complicar con auth compleja (basta auth básica).
- Usar scripts de seeds automáticos para generar 500 páginas de prueba si no hay seeds reales: crear micro-site estático en otro contenedor que sirva páginas.