AI-Powered Content Summarizer & Researcher

Tecnical Design Document

Gabriel de Almeida Miki

Sumário

Introduction

The Intelligent Research Hub is a containerized, full-stack web application designed to automate the process of information gathering and synthesis. By leveraging Django for management and Google Agent Kit for intelligent reasoning, the platform allows users to transform long-form articles, research papers, or web URLs into concise, actionable summaries and structured data.
The application is built with a microservices-oriented architecture, using Docker to decouple the web interface, the background task processor, and the data persistence layers. This ensures that heavy AI processing tasks do not block the user interface, providing a seamless, "asynchronous" research experience.

Functional

- URL & Text Processing: Users must be able to input a raw URL or paste plain text into a dashboard for analysis.
- AI Research Agent: The system must utilize the Google Agent Kit to browse the provided content, identify key themes, and generate a summary.
- Asynchronous Task Management: Long-running AI tasks must be offloaded to a Celery worker via a Redis broker to prevent request timeouts.
- Data Persistence: All generated summaries, original sources, and research timestamps must be stored in a PostgreSQL database.
- Research History Dashboard: Users must have a secure dashboard to view, search, and delete their previous research results.
- User Authentication: A secure login system to ensure users can only access their own research data.

Non Functional

- Scalability (Orchestration): The application must be orchestrated using Docker Compose, allowing services (Web, Worker, DB) to scale independently.
- Optimized Build Performance: The production Docker image must use Multi-Stage builds to ensure the final image size is under 300MB, excluding build-time-only dependencies.
- Deployment Efficiency (Caching): The Dockerfile must be structured to leverage Layer Caching, ensuring that code changes do not trigger a full re-installation of heavy Python libraries.
- High Availability: The database must use Docker Volumes to ensure data persistence across container restarts or failures.
- Security: Sensitive information, such as Google API keys and database credentials, must be managed via Environment Variables and never hardcoded into the image layers.

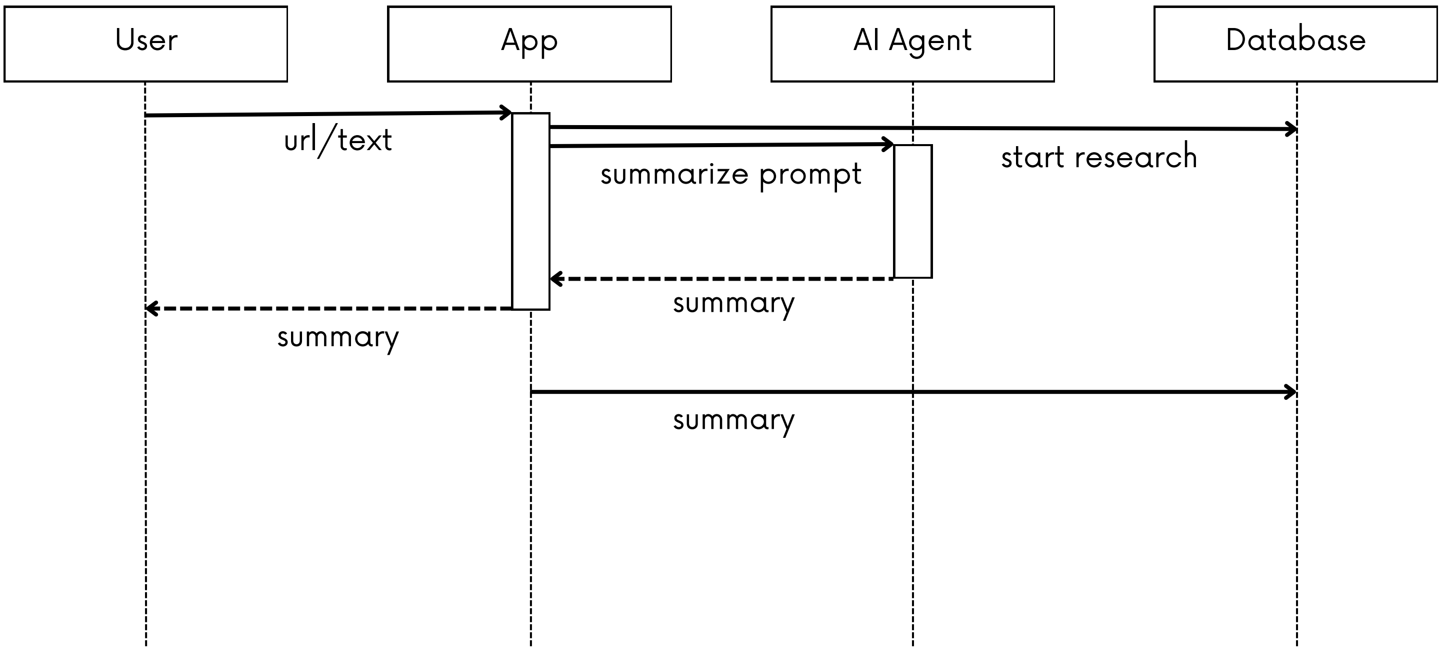
- Environment Parity: The Docker configuration must ensure that the application runs identically in development and production environments by isolating all system-level dependencies.

Use Case: Submit Research Task

- **Actors**:
  - **User**: The person providing a source for analysis.
  - **Google Agent Service**: The AI logic processing the content.
  - **Task Queue (Redis/Celery)**: The system managing the background execution.
- **Entry condition**:
  - The user is authenticated and on the "New Research" dashboard page.
- **Flow of Events**:
  - The user enters a URL or pastes text into the submission form.
  - The user clicks "Start Research."
  - The Django application validates the input and creates a "Pending" record in the PostgreSQL database.
  - The application pushes a job to the Redis queue.
  - A Celery Worker picks up the job and invokes the Google Agent Kit.
  - The agent scrapes, analyzes, and summarizes the content.
  - The worker updates the database record status to "Completed" and saves the summary.
- **Exit condition**:
  - The task is successfully queued, and the user sees a "Processing" notification.
- **Exceptions**:
  - **Invalid Source**: The URL is unreachable or the text is empty; the user is prompted to correct the input.
  - **AI Rate Limiting**: The Google Agent Kit hits an API limit; the task is retried automatically by the worker or flagged as failed.
- **Special Requirements**:
  - Input sanitization to prevent injection attacks.
  - The worker must be able to handle "headless" browsing if the URL requires JavaScript rendering.
- **Nonfunctional Requirements, Constraints**:
  - The UI must remain responsive while the AI processes (Asynchronous handling).
  - The hand-off from Django to Redis must occur in under 200ms.

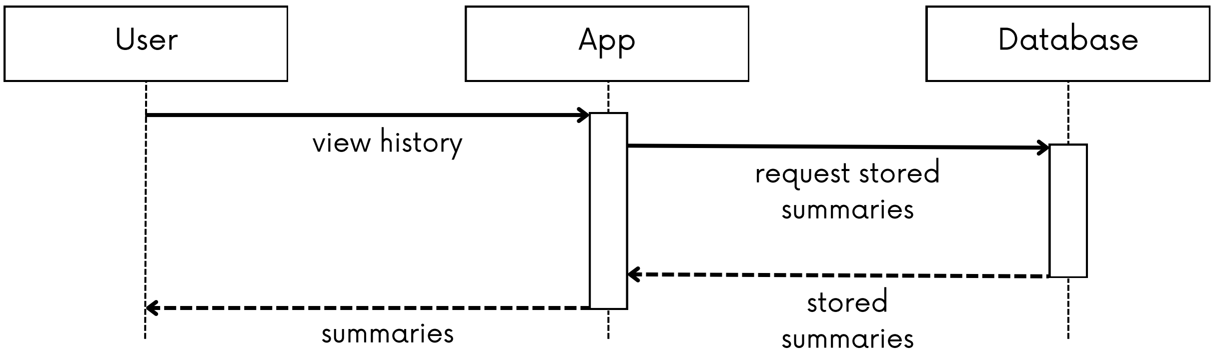
Use Case: View and Manage Research History

- **Actors**:
  - **User**: The person reviewing past work.
  - **Database (PostgreSQL)**: The persistent storage for research data.
- **Entry condition**:
  - The user is logged in and navigates to the "My Research" archive.
- **Flow of Events**:
  - The application queries the database for all records associated with the User ID.
  - The system displays a list of summaries with timestamps and source titles.
  - The user selects a specific entry to view the full AI-generated report.
  - (Optional) The user clicks "Delete" to remove an entry.
  - The system updates the database and refreshes the view.
- **Exit condition**:
  - The user has viewed or modified their research history.
- **Exceptions**:
  - **Database Timeout**: If the database is under heavy load, a "Service Unavailable" message is shown.
  - **No Records Found**: The user is shown a "Get Started" prompt if their history is empty.
- **Special Requirements**:
  - Pagination for users with a large number of research entries.
  - Search and filter functionality (e.g., filter by date or keywords).
- **Nonfunctional Requirements, Constraints**:
  - The history list must load in under 1 second.
  - Data must be persistent (survive container restarts) via Docker Volumes.



Use Case: Export Research Report

- **Actors**:
  - **User**: The person wanting a portable version of the research.
  - **Export Service**: A Django utility that converts database records into PDF or Markdown.
- **Entry condition**:
  - The user is viewing a "Completed" research entry.
- **Flow of Events**:
  - The user clicks the "Export as PDF" button.
  - The Django application fetches the summary, source metadata, and AI insights from PostgreSQL.
  - The system passes this data to a document-rendering engine (e.g., ReportLab or WeasyPrint).
  - The application generates a temporary file in memory.
  - The browser triggers a download for the user.
- **Exit condition**:
  - The user successfully downloads the .pdf file.
- **Exceptions**:
  - **Missing Dependencies**: If the Docker image is missing system-level PDF libraries (like pango), the export fails. (This is a great test for your Image Building skills).
- **Nonfunctional Requirements, Constraints**:
  - Export generation should take less than 5 seconds.
  - The Docker image must include the necessary C-libraries for PDF rendering without bloating the final image size (requires Multi-Stage Build optimization).

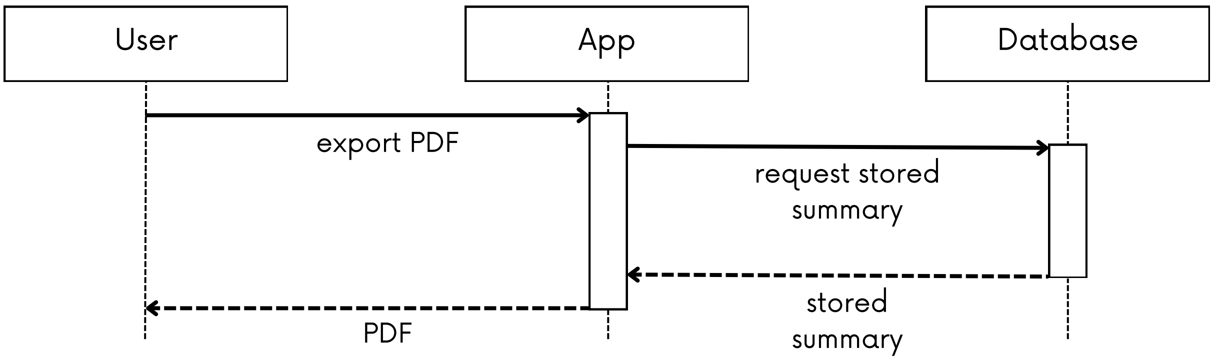


Use Case: Real-time Progress Tracking

- **Actors**:
  - **User**: The person waiting for results.
  - **Websocket Server (Django Channels)**: The service pushing live updates.
  - **Celery Worker**: The process reporting its progress.

Use Case: Real-time Progress Tracking

- **Entry condition**:
  - A research task has just been submitted and is in the "Processing" state.
- **Flow of Events**:
  - The user is redirected to a "Live Progress" page.
  - The browser opens a WebSocket connection to the Django container.
  - As the Google Agent works, the Celery worker sends updates: "Scraping URL...", "Analyzing Sentiment...", "Generating Summary...".
  - The backend broadcasts these messages to the user's dashboard in real-time.
- **Exit condition**:
  - The progress bar reaches 100%, and the full summary is revealed.
- **Special Requirements:**
  - Requires an ASGI server (like Daphne or Uvicorn) inside your Docker container instead of a standard WSGI server.
- **Nonfunctional Requirements, Constraints**:
  - Latency for status updates should be less than 500ms.

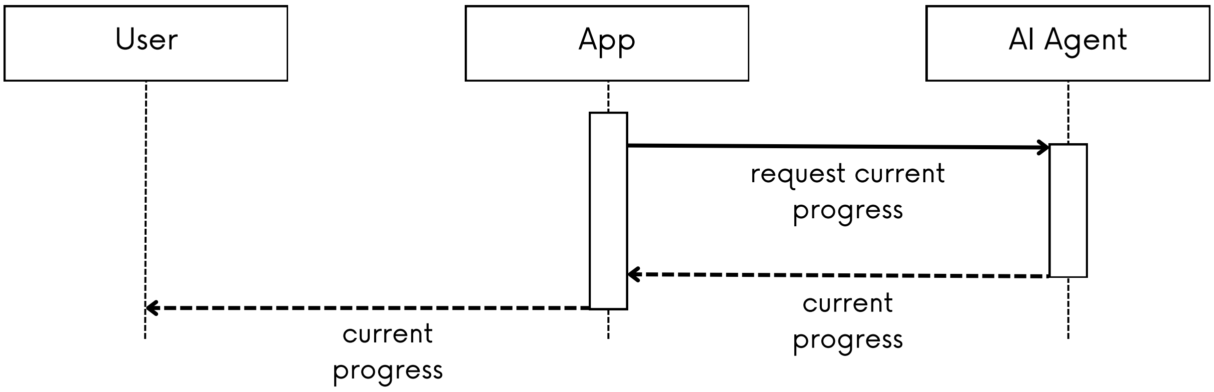


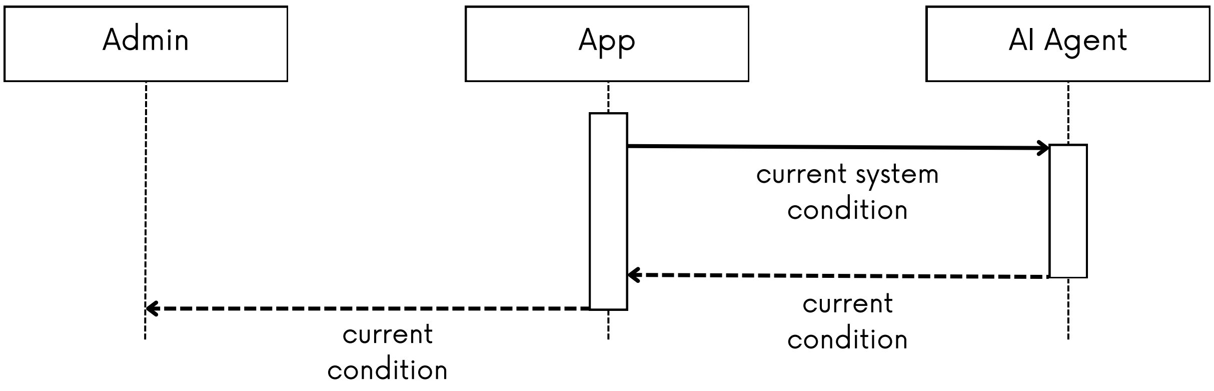
Use Case: System Health Monitoring (Admin)

- **Actors**:
  - **Admin**: The developer (you) monitoring the system.
  - **Docker Engine**: The underlying container manager.
- **Entry condition**:
  - The Admin accesses the hidden /admin/health endpoint or a dedicated monitoring dashboard.
- **Flow of Events**:
  - The system queries the status of the Redis broker and PostgreSQL database.
  - The application checks the "heartbeat" of the active Celery Workers.
  - The dashboard displays the CPU and Memory usage of each container.
  - The Admin reviews logs to ensure the Google Agent Kit is not encountering authentication errors.

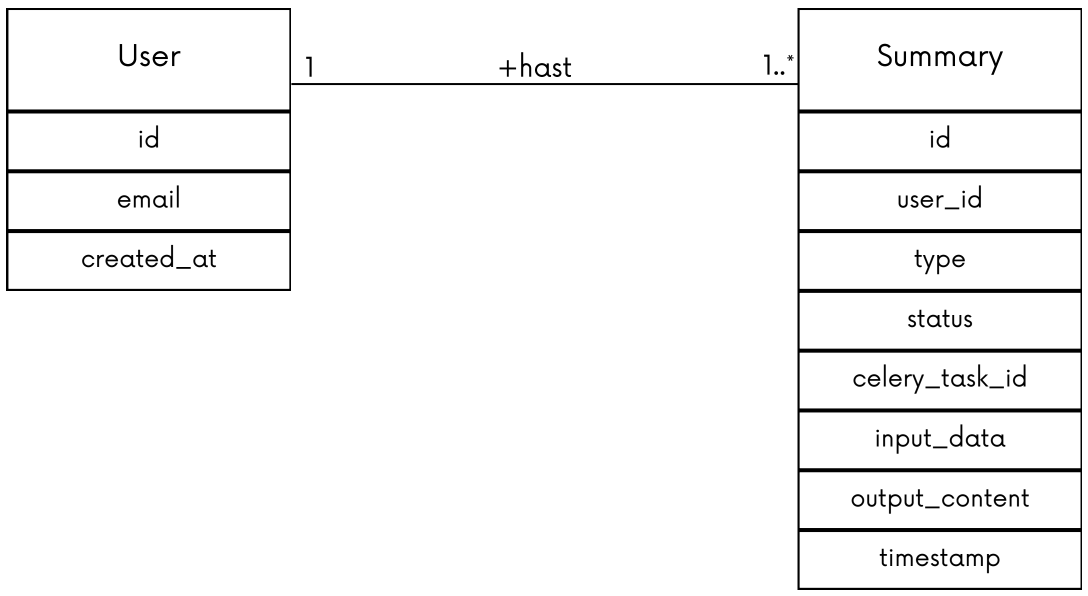
Use Case: System Health Monitoring (Admin)

- **Exit condition**:
  - The Admin confirms all services are healthy.
- **Exceptions**:
  - **Service Down**: If Redis is unreachable, the dashboard highlights the "Worker" as disconnected.
- **Nonfunctional Requirements, Constraints**:
  - The monitoring tool should have minimal overhead on the production containers.
  - Health checks must be automated using the HEALTHCHECK instruction in the Dockerfile.





Database Design

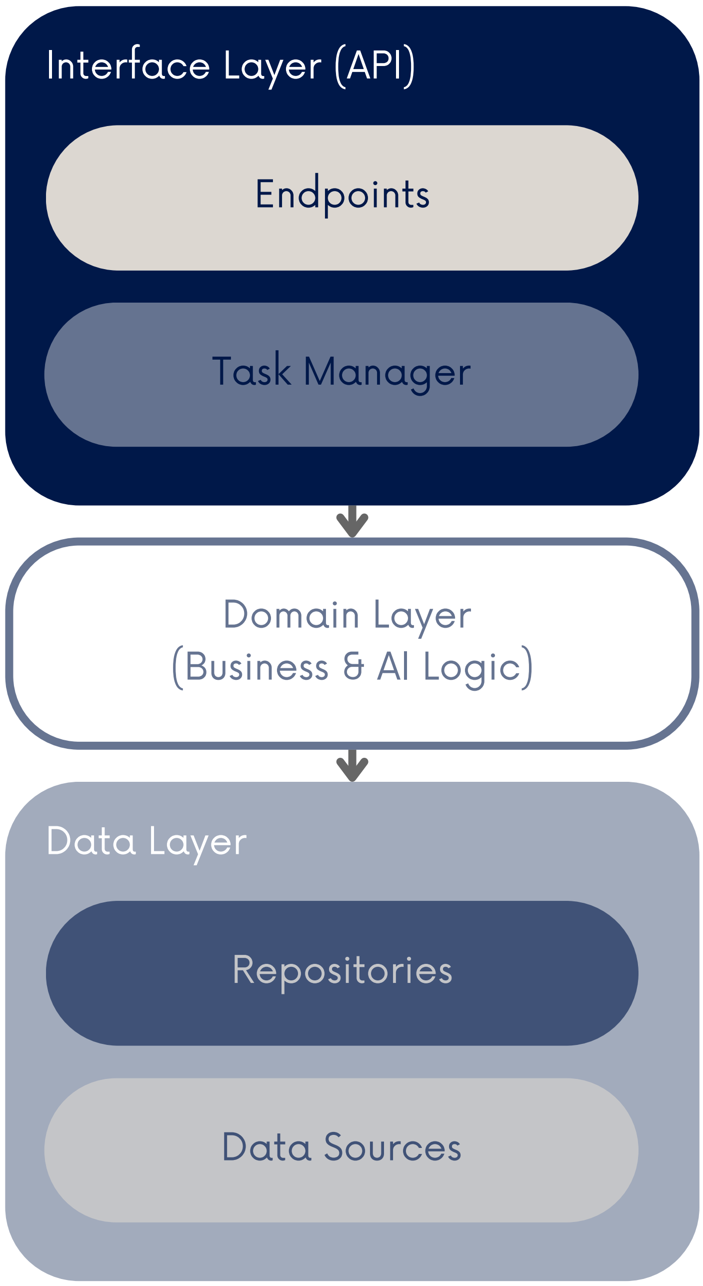


Entity: Task / Summary
Each research or summary input is detailed with metadata

- id (PK)
- user_id (FK) - ties to user.
- type - classifies weather it is a "Summary" or a "Research Report".
- status - essential to Celery: 'pending', 'processing', 'completed', 'failed'.
- celery_task_id - in order to query the progress.
- input_data - URL or original text sent.
- output_content - final AI result.
- timestamp - date and time of creation.

Technical Design

The application is structured into three primary layers: the Interface (API) layer, the Domain (Business & AI) layer, and the Data and Infrastructure layer. This layered architecture ensures a clear separation of concerns, promoting maintainability, scalability (especially for background workers), and testability across all components.



Interface Layer (API)

The API layer serves as the entry point for the application, responsible for receiving user requests and providing immediate responses. In the context of an asynchronous architecture using Celery, it does not perform heavy processing but instead manages the communication flow. This layer comprises:

- Endpoints/Controllers: These manage HTTP routes (e.g., /summarize) and validate input data such as URLs, text, or research queries.
- Task Producers: Classes responsible for packaging user requests and dispatching them to the Redis queue.
- Task State Managers: Mechanisms that expose the processing status (pending, completed, or error) to the frontend via polling or WebSockets.

The primary functions of the Interface layer include:

- Validating and sanitizing user inputs.
- Orchestrating the creation of asynchronous tasks in Redis.
- Managing user authentication and authorization.

Domain Layer (Business & AI Logic)

The domain layer serves as the "brain" of the application, housing the business logic that distinguishes it from a simple database. It remains agnostic regarding how data is stored or how the API is exposed. This layer comprises:

- Celery Workers: Components that execute heavy-duty logic outside the standard API request-response cycle.
- AI Engine (Wrappers): Modules that manage communication with LLMs (Large Language Models), handling prompt engineering and natural language processing.
- Researcher Logic: Algorithms that determine how to fetch external information, filter results, and synthesize complex reports.

The primary functions of the Domain layer include:

- Transforming raw data (long articles) into structured information (summaries).
- Ensuring that research and synthesis logic remains independent of specific AI providers.
- Executing long-running tasks in a resilient and isolated manner.

Data Layer (Persistence & Infrastructure)

The data layer contains the persistence logic and the communication infrastructure between different parts of the application.

It defines how information is stored, retrieved, and moved between the backend and the workers. This layer is composed of:

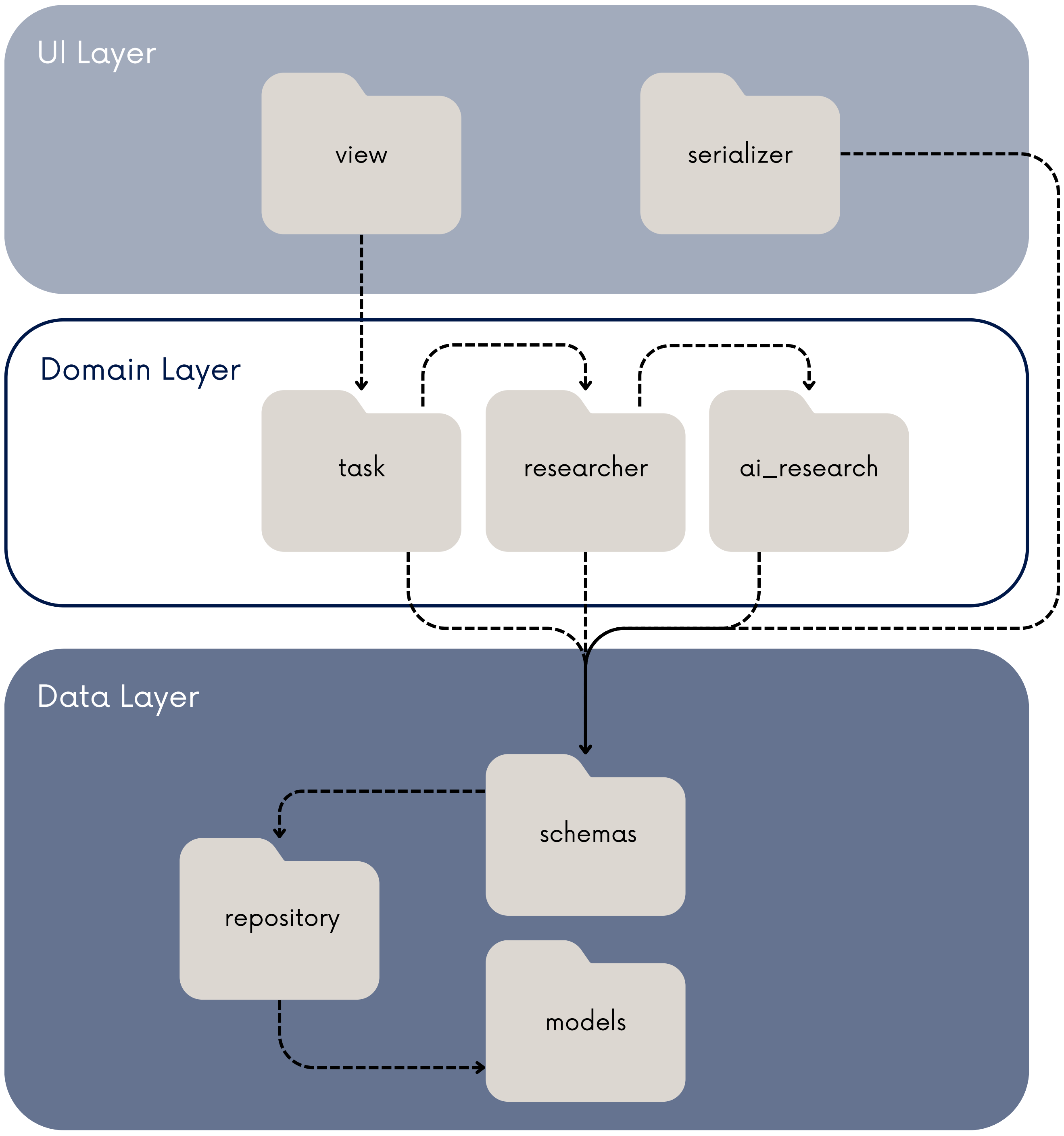
- Postgres Repositories: Classes responsible for centralizing access to Postgres, managing CRUD operations for users and summary history.
- Message Broker (Redis): Manages the task queue, ensuring data flows from the API to the Workers without loss.
- Data Models (Core): Data schema definitions (such as Pydantic models or Dataclasses) that unify the information format across all layers.

The primary functions of the Data layer include:

- Data Exposure: Providing access to summary history and user profiles.
- Persistence Management: Centralizing and saving the output generated by the AI.
- Infrastructure Abstraction: Isolating the complexity of database and message broker connections from the rest of the application.

Package Diagram

The package diagram provides a high-level view of the application's architecture. It illustrates how the application is organized into distinct layers and modules, depicting the dependencies and relationships between these packages. This diagram helps in understanding the separation of concerns and the overall structure of the system.



Development - Docker Principles

Industry Best Practices Applied Here

- Stateless Web Containers: Your Django containers can be destroyed and recreated at any time without data loss because all state is in Postgres (Data) or Redis (Cache/Queue).

Container Group	**Service Name**	**Role & Best Practice Note**
The Gatekeeper	Nginx (Reverse Proxy)	Entry Point. It sits in front of Django cluster. It handles SSL termination (HTTPS), serves static files (CSS/JS) efficiently, and load-balances requests to the available Django containers.
The Backend Cluster	Django (API Service)	Stateless. These containers handle the HTTP requests. You can spin up 3, 5, or 10 of these depending on user traffic. They never store data locally; they only talk to Postgres or Redis.
The Brain Cluster	Celery Workers (AI Agents)	The Heavy Lifters. This is where "AI Agents" live. They listen to Redis queue. Unlike the web containers, these can take 30+ seconds to process a request without blocking the user interface. You can scale these based on queue depth (how many tasks are waiting).
The Broker	Redis	Centralized Broker. The glue between Django and Celery. It holds the "To-Do" list. In production, this can be a single container with persistence enabled or a managed cloud service.
The Vault	PostgreSQL	Centralized Data. Stores Users and Summaries. Crucial: In a containerized setup, it must mount a Docker Volume to this container so data survives if the container crashes or restarts.

- Single Process per Container:
    - Don't try to run Django AND Celery in the same container. Keep them separate. This allows you to scale them differently (e.g., if you have few users but heavy AI usage, you add more Celery containers, not Django ones).
- Volumes for Persistence: The Database and Redis containers must use mapped volumes (e.g., -v pgdata:/var/lib/postgresql/data) to persist data on the host machine.
- Internal Networking:
    - Public Network: Only Nginx exposes ports (80/443) to the outside world.
    - Private Network: Django, Postgres, Redis, and Celery talk to each other on a private internal Docker network. The database port (5432) is never exposed to the public internet.

```yaml
version: '3.8'

services:
  # --- The Gatekeeper ---
  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    depends_on:
      - web

  # --- The Backend Cluster ---
  web:
    build: .
    command: gunicorn core.wsgi:application --bind 0.0.0.0:8000
    deploy:
      replicas: 3  # Running 3 instances of Django
    env_file: .env
    depends_on:
      - db
      - redis

  # --- The AI Agent Cluster ---
  ai_worker:
    build: .
    command: celery -A core worker -l info -Q ai_queue
    deploy:
      replicas: 2 # Running 2 AI agents
    env_file: .env
    environment:
      - GOOGLE_APPLICATION_CREDENTIALS=/app/credentials.json
    depends_on:
      - redis

  # --- Centralized Services ---
  db:
    image: postgres:15
    volumes:
      - postgres_data:/var/lib/postgresql/data # Persistence

  redis:
    image: redis:alpine

volumes:
  postgres_data:
```

Analysing Proposed Architecture

The proposed architecture stipulates running everything (including DB and Redis) in containers, likely orchestrated by Docker Compose or a single-node setup.

- Why this is GOOD:
  - Dev/Prod Parity: What runs on your laptop is exactly what runs on the server.
  - Simplicity: control everything in one file (docker-compose.yml).
  - Cost: can run this whole stack on a single VPS (e.g., one AWS EC2 instance or DigitalOcean Droplet) for S20/month.

- Why this is RISKY (The Industry Critique):
  a. The "Stateful Container" Problem: Running a database (Postgres) inside a container in production is risky. If the container crashes or the file system corrupts, recovering data is harder than using a managed service.
  b. Single Point of Failure: If the machine hosting your "cluster of containers" dies, your entire app (DB, API, AI) vanishes.
  c. Scaling Limitation: Docker Compose does not natively support "Auto-Scaling" across multiple machines. If the AI traffic spikes, it is not easily possible to add more servers just for the workers without manual intervention.

The "Better" Architecture: Cloud-Native Hybrid

To align with the highest industry standards for robustness and scalability, it would be necessary to move from a "Containerized Monolith" to a "Cloud-Native Hybrid Architecture."

The key change is "Decoupling State from Compute." The Changes Explained:

1. Database ➡️ Managed Service (RDS / Cloud SQL)
   - Change: Instead of a postgres container, use a Cloud Provider's database (AWS RDS, Google Cloud SQL, Azure SQL).
   - Justification: Automated backups, point-in-time recovery, and high availability (Multi-AZ). You don't manage the OS patches for the DB.
2. Redis ➡️ Managed Cache (ElastiCache / Memorystore)
   - Change: Use a managed Redis instance.
   - Justification: If Redis container restarts, it loses the queue of pending AI tasks. Managed Redis ensures persistence and prevents task loss during updates.

The "Better" Architecture: Cloud-Native Hybrid

3. Static Files ➡️ Object Storage (S3 / GCS)
    - Change: Nginx should not serve static files (images, CSS) from a local folder. Django should upload them to AWS S3 or Google Cloud Storage.
    - Justification: This allows your Django containers to be totally ephemeral. You can destroy and recreate them instantly without worrying about losing user-uploaded avatars or reports.
4. AI Orchestration ➡️ Horizontal Autoscaling
    - Change: Use a container orchestrator (Kubernetes or AWS ECS/Fargate) specifically for the Celery Workers.
    - Justification: AI tasks are resource-intensive. With this setup, you can set a rule: "If Redis queue > 50 items, add 10 more Worker Containers automatically."

```yaml
version: '3.8'

services:
  # ---------------------------------------------------------------------------
  # REVERSE PROXY (The Gatekeeper)
  # ---------------------------------------------------------------------------
  nginx:
    image: nginx:1.25-alpine
    container_name: finglance_nginx
    ports:
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf:ro
      - static_volume:/app/static # Serve static files directly
    depends_on:
      - web
    networks:
      - public_network
      - internal_network

  # ---------------------------------------------------------------------------
  # WEB SERVER (Django + Gunicorn)
  # ---------------------------------------------------------------------------
  web:
    build: .
    container_name: finglance_web
    # Wait for DB, migrate, collect static files, then start Gunicorn
    command: >
      sh -c "python manage.py migrate &&
             python manage.py collectstatic --noinput &&
             gunicorn core.wsgi:application --bind 0.0.0.0:8000"
    volumes:
      - .:/app  # Hot-reloading for development (remove in prod)
      - static_volume:/app/static # Share static files with Nginx
      - ./credentials.json:/app/credentials.json:ro # Google Auth
    env_file:
      - .env
    depends_on:
      db:
        condition: service_healthy
      redis:
        condition: service_started
    networks:
      - internal_network

  # ---------------------------------------------------------------------------
  # AI AGENT WORKER (Celery)
  # ---------------------------------------------------------------------------
  worker:
    build: .
    container_name: finglance_worker
    command: celery -A core worker -l info -Q default
    volumes:
      - .:/app
      - ./credentials.json:/app/credentials.json:ro # Google Auth
    env_file:
      - .env
    depends_on:
      - redis
      - db
```

Improving security and bottlenecks

The first goal is to separate the "Build" environment (compilers, headers) from the "Runtime" environment. This removes vulnerabilities and drastically reduces image size.

```
# ---------------------------------------------
# Stage 1: The Builder (Compilers & Build Tools)
# ---------------------------------------------
# Pinning version for stability (Section 1.2)
FROM python:3.11.4-slim-bookworm as builder

WORKDIR /app

# Consolidating layers (Section 1.2)
RUN apt-get update && apt-get install -y --no-install-recommends \
    build-essential \
    libpq-dev \
    && rm -rf /var/lib/apt/lists/*

# Install dependencies to a virtual environment
RUN python -m venv /opt/venv
# Ensure we use the venv
ENV PATH="/opt/venv/bin:$PATH"

COPY requirements.txt .
RUN pip install --no-cache-dir --upgrade pip && \
    pip install --no-cache-dir -r requirements.txt

# -----------------------------------------------------------------------------
# Stage 2: The Final Runtime (Minimal & Secure)
# -----------------------------------------------------------------------------
FROM python:3.11.4-slim-bookworm

# Create a non-root user first (Section 1.3)
RUN groupadd -r appgroup && useradd -r -g appgroup -s /sbin/nologin appuser

WORKDIR /app

# Install ONLY runtime libraries (e.g. libpq5, not libpq-dev)
RUN apt-get update && apt-get install -y --no-install-recommends \
    libpq5 \
    netcat-openbsd \
    curl \
    && rm -rf /var/lib/apt/lists/*

# Copy the virtual environment from the builder stage
COPY --from=builder /opt/venv /opt/venv

# Set environment variables
ENV PATH="/opt/venv/bin:$PATH"
ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1

# Copy application code (Layer Ordering: Code changes most often, so it's last)
COPY . /app/
# Ensure the non-root user owns the files
RUN chown -R appuser:appgroup /app

# Switch to non-root user
USER appuser

# Documentation port
EXPOSE 8000
```

Improving security and bottlenecks

Now, instead of passing POSTGRES_PASSWORD=secret in plain text via .env, we will mount a file securely into /run/secrets/.

```yaml
services:
  web:
    # ... (other settings)
    secrets:
      - db_password
      - django_secret_key
      - google_credentials
    environment:
      # We tell the app WHERE the secret is, not WHAT it is (Convention)
      - DB_PASSWORD_FILE=/run/secrets/db_password
      - GOOGLE_APPLICATION_CREDENTIALS=/run/secrets/google_credentials

  db:
    image: postgres:15-alpine
    secrets:
      - db_password
    environment:
      - POSTGRES_PASSWORD_FILE=/run/secrets/db_password

secrets:
  db_password:
    file: ./secrets/db_password.txt  # This file should NOT be in git
  django_secret_key:
    file: ./secrets/django_key.txt
  google_credentials:
    file: ./credentials.json
```

Crucial Code Change (settings.py): Since Django doesn't read _FILE suffixes by default, you need this utility in your core/settings.py to comply with the guide's pattern:

```python
import os

def get_secret(secret_name, default=None):
    """
    Reads the secret from a file (Docker Secret) if the _FILE env var exists,
    otherwise falls back to the standard env var.
    """
    file_path = os.environ.get(f"{secret_name}_FILE")
    if file_path:
        with open(file_path, 'r') as f:
            return f.read().strip()
    return os.environ.get(secret_name, default)

# Usage
SECRET_KEY = get_secret("DJANGO_SECRET_KEY")
DATABASES = {
    'default': {
        'PASSWORD': get_secret("DB_PASSWORD"),
        # ... other settings
    }
}
```

Improving security and bottlenecks

It is also interesting to add explicit memory/cpu limits and robust healthchecks.

```
web:
    # ...
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health/"]
      interval: 30s
      timeout: 10s
      retries: 3
    deploy:
      resources:
        limits:
          cpus: '0.50'        # Use at most 50% of 1 CPU core
          memory: 512M        # Kill container if it exceeds 512MB
        reservations:
          memory: 256M        # Guarantee 256MB
    restart: always           # Section 4.3

  worker:
    # ...
    deploy:
      resources:
        limits:
          cpus: '1.0'         # AI tasks are heavy, allow 1 full core
          memory: 1G          # Allow more RAM for AI processing
    restart: always
```

Needed Libraries

Here is a brief breakdown of each library and its specific role in the architecture:

Core Framework

- **Django**: The high-level Python web framework. It handles the "heavy lifting" like URL routing, database management (ORM), and authentication.
    - Usage: Serves as the backbone of the backend application.
- **djangorestframework (DRF)**: A toolkit built on top of Django for building Web APIs.
    - Usage: Transforms Django models into JSON data that your frontend can understand (Serialization).
- **django-cors-headers**: A Django extension that adds Cross-Origin Resource Sharing (CORS) headers to responses.
    - Usage: Allows a frontend (like React or Vue) running on a different domain/port (e.g., localhost:3000) to successfully make requests to the API (e.g., localhost:8000).

Application Server

- **uvicorn**: A lightning-fast ASGI (Asynchronous Server Gateway Interface) server.
    - Usage: Actually runs the Python code, handling asynchronous connections (WebSockets or async views) efficiently.
- **gunicorn**: A WSGI HTTP Server for UNIX. It acts as a process manager.
    - Usage: Sits "on top" of Uvicorn in production. It spawns and manages multiple "workers" (copies of your app) to handle concurrent requests and restart them if they crash.

Database

- **psycopg2-binary**: The most popular PostgreSQL adapter for Python.
    - Usage: Allows Django to "talk" to your PostgreSQL database container to save and retrieve data.

Async Task Queue

- **celery**: A distributed task queue system.
    - Usage: Offloads heavy tasks (like "Summarize this 50-page PDF") from the user's request loop to a background worker, preventing the website from freezing.
- **redis**: An in-memory data structure store.
    - Usage: Acts as the "Message Broker" for Celery. Django puts a task in Redis, and Celery workers constantly check Redis to pick up new work.
- **flower**: A real-time web-based monitoring tool for Celery.
    - Usage: Provides a visual dashboard to see how many tasks are running, how many failed, and how fast your workers are processing data.

Configuration & Security

- **django-environ**: A library to manage configuration via environment variables.
    - Usage: Reads your .env file (where secrets like DB_PASSWORD are stored) and automatically configures the Django settings.py. It creates a clean 12-Factor App compliant setup.

API Documentation

- **drf-spectacular**: An OpenAPI schema generator for Django Rest Framework.
    - Usage: Automatically scans the code and generates a professional, interactive "Swagger UI" documentation page (/api/schema/swagger-ui/) where developers can test your API endpoints.

AI & Research

- **google-cloud-aiplatform**: The official Google Cloud Client Library for Vertex AI.
    - Usage: The bridge between your Python code and Google's powerful AI models (Gemini) and Agent Builder. This is how you send prompts and receive summaries.
- **google-adk (Optional)**: The Agent Development Kit.
    - Usage: Provides specialized tools for building "Agentic" workflows if you decide to build custom agents rather than just consuming Vertex AI endpoints.
- **requests**: A simple HTTP library for Python.
    - Usage: Used by your "Researcher" logic to fetch raw HTML from websites if the AI agent needs to read a specific article directly.
- **beautifulsoup4**: A library for pulling data out of HTML and XML files.
    - Usage: Cleans up the "messy" HTML fetched by requests (removing ads, menus, scripts) so the AI only reads the relevant article text.

API Specification v1.0

Base URL: /api/v1

Authentication: Session Authentication (Browser) or Token Authentication (Header: Authorization: Token <token>)

Content-Type: application/json

Resources

Field	Type	Description
id	UUID	Unique identifier for the summary task.
url	URL	The target article URL to summarize.
status	String	Current state: PENDING, PROCESSING, COMPLETED, FAILED.
title	String	The detected title of the article (optional, populated after scrape).
summary	Text	The AI-generated summary (null until status is COMPLETED).
created_at	DateTime	ISO 8601 timestamp of creation.

Submit New Research Task

Initiates the asynchronous background process. The server returns immediately with 202 Accepted to prevent blocking.

- Endpoint: POST /summaries/
- Access: Authenticated Users

Request Body:

```
"url": "https://www.bloomberg.com/news/articles/2026-02-01/market-trends"
```

Response: 202 Accepted (Task successfully queued)

```
{
  "id": "a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11",
  "url": "https://www.bloomberg.com/news/articles/2026-02-01/market-trends",
  "status": "PENDING",
  "created_at": "2026-02-01T10:00:00Z",
  "message": "Research task started successfully."
}
```

Response: 400 Bad Request (Invalid URL)

```
{
  "url": ["Enter a valid URL."]
}
```

Check Task Status / Retrieve Result

Poll this endpoint to check if the AI has finished.

- Endpoint: GET /summaries/{id}/
- Access: Owner Only (User who created the task)

Response: 200 OK (Still Processing)

```
{
  "id": "a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11",
  "status": "PROCESSING",
  "created_at": "2026-02-01T10:00:00Z",
  "summary": null
}
```

Check Task Status / Retrieve Result

Response: 200 OK (Completed)

```
{
  "id": "a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11",
  "status": "COMPLETED",
  "title": "Global Markets Rally as Tech Stocks Surge",
  "summary": "## Executive Summary\n\nGlobal markets saw a significant uptick today driven by...",
  "created_at": "2026-02-01T10:00:00Z",
  "completed_at": "2026-02-01T10:00:45Z"
}
```

List User History

Retrieves a paginated list of all research tasks performed by the current user.

- Endpoint: GET /summaries/
- Access: Authenticated Users

Query Parameters:

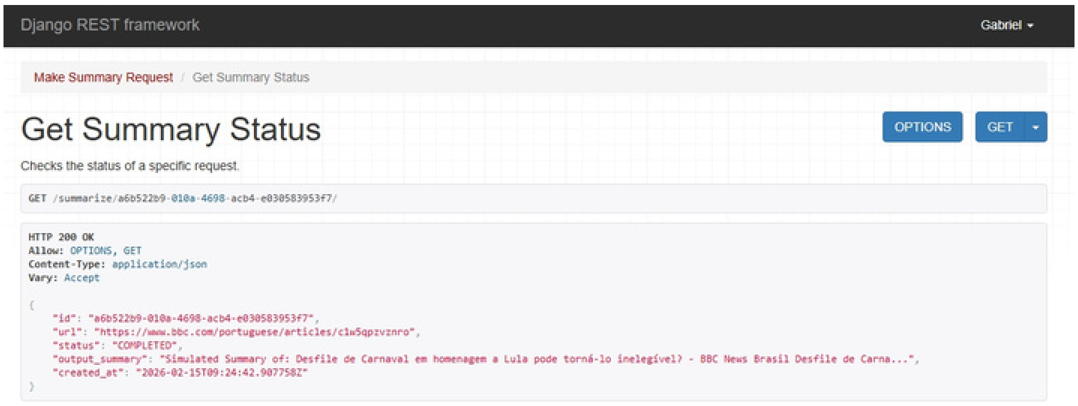
- page: Page number (default: 1)
- status: Filter by status (e.g., ?status=COMPLETED)

Response: 200 OK

```
{
  "count": 52,
  "next": "http://api.finglance.com/api/v1/summaries/?page=2",
  "previous": null,
  "results": [
    {
      "id": "a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11",
      "url": "https://www.bloomberg.com/...",
      "status": "COMPLETED",
      "title": "Global Markets Rally...",
      "created_at": "2026-02-01T10:00:00Z"
    },
    {
      "id": "b1ffcd22-1d1c-5fa9-cc7e-7cc0ce491b22",
      "url": "https://techcrunch.com/...",
      "status": "FAILED",
      "title": null,
      "created_at": "2026-01-31T09:30:00Z"
    }
  ]
}
```

Development Steps

After testing and verifying unit and combined functionality, the application was launched.



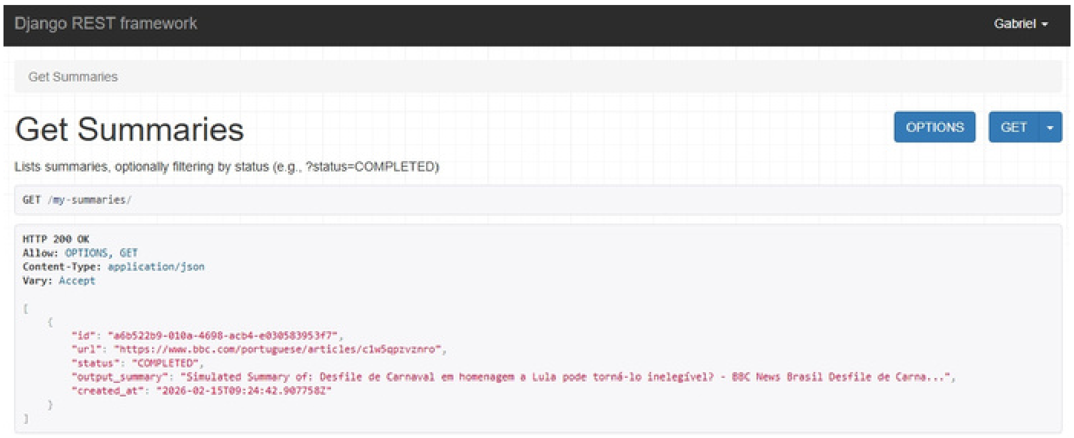
Get Summary Status

Checks the status of a specific request.

```
GET /summarize/a6b522b9-010a-4698-acb4-e030583953f7/
```

```
HTTP 200 OK
Allow: OPTIONS, GET
Content-Type: application/json
Vary: Accept

{
    "id": "a6b522b9-010a-4698-acb4-e030583953f7",
    "url": "https://www.bbc.com/portuguese/articles/c1w5qpzvznro",
    "status": "COMPLETED",
    "output_summary": "Simulated Summary of: Desfile de Carnaval em homenagem a Lula pode torná-lo inelegível? - BBC News Brasil Desfile de Carna...",
    "created_at": "2026-02-15T09:24:42.907758Z"
}
```



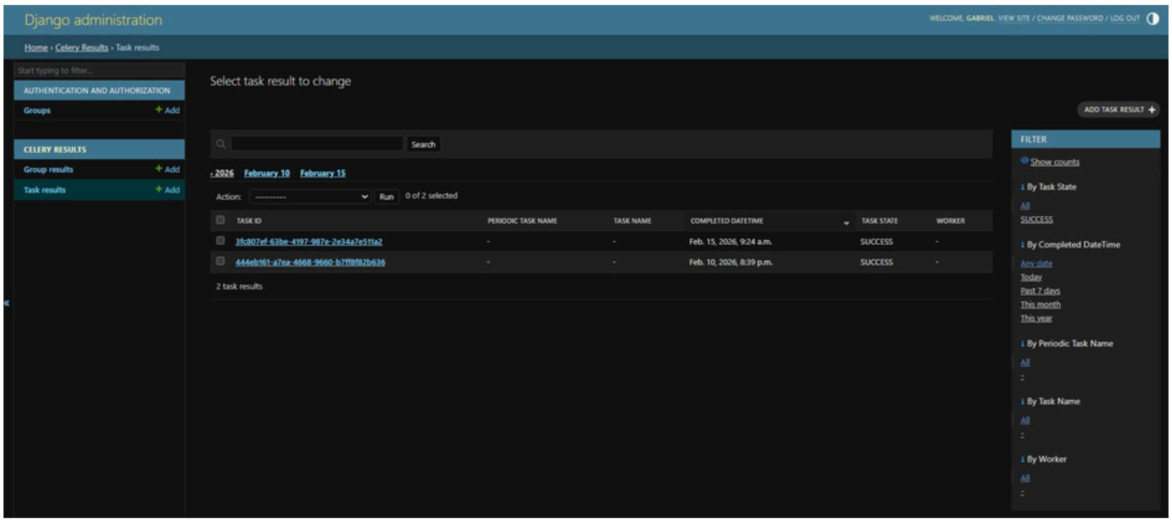
Get Summaries

Lists summaries, optionally filtering by status (e.g., ?status=COMPLETED)

```
GET /my-summaries/
```

```
HTTP 200 OK
Allow: OPTIONS, GET
Content-Type: application/json
Vary: Accept

[
    {
        "id": "a6b522b9-010a-4698-acb4-e030583953f7",
        "url": "https://www.bbc.com/portuguese/articles/c1w5qpzvznro",
        "status": "COMPLETED",
        "output_summary": "Simulated Summary of: Desfile de Carnaval em homenagem a Lula pode torná-lo inelegível? - BBC News Brasil Desfile de Carna...",
        "created_at": "2026-02-15T09:24:42.907758Z"
    }
]
```

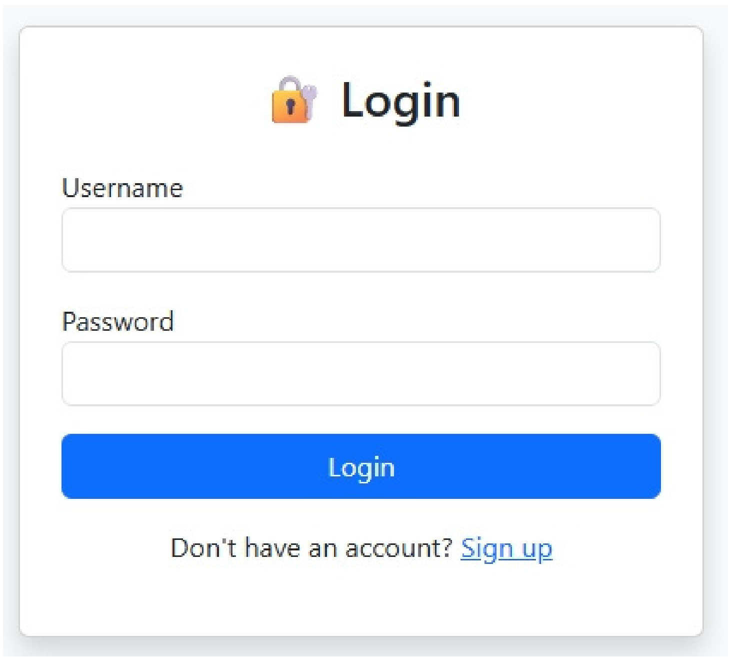


Select task result to change

TASK ID	PERIODIC TASK NAME	TASK NAME	COMPLETED DATETIME	TASK STATE	WORKER
3fc807ef-63be-4197-987e-2e34a7e511a2	-	-	Feb. 15, 2026, 9:24 a.m.	SUCCESS	-
444eb161-a7ea-4668-9660-b7ff8f82b636	-	-	Feb. 10, 2026, 8:39 p.m.	SUCCESS	-

2 task results

After confirming it was working correctly, through the usage of AI tools, a frontend was added to the application.

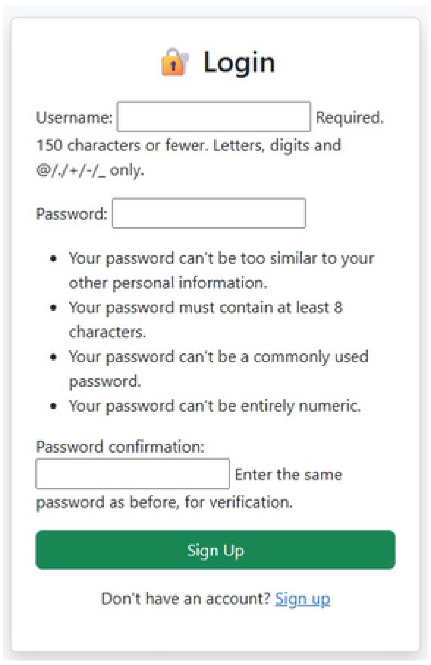

🔐 Login

Username:

Password:

Login

Don't have an account? Sign up


🔐 Login

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

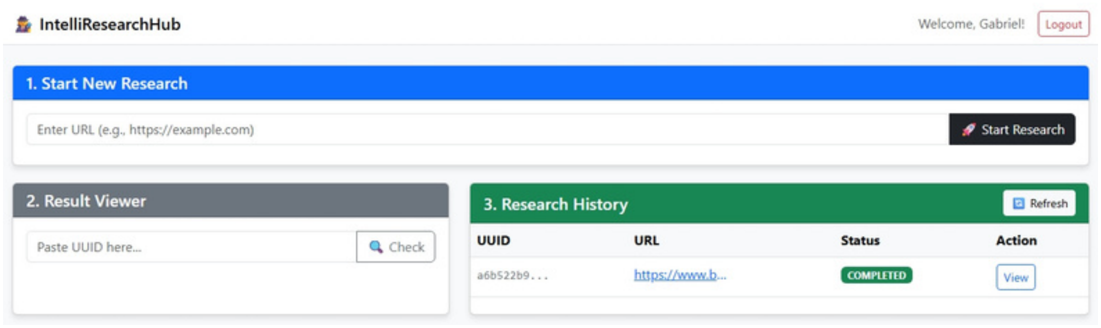
Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

Sign Up

Don't have an account? Sign up


IntelliResearchHub — Welcome, Gabriel! Logout

1. Start New Research

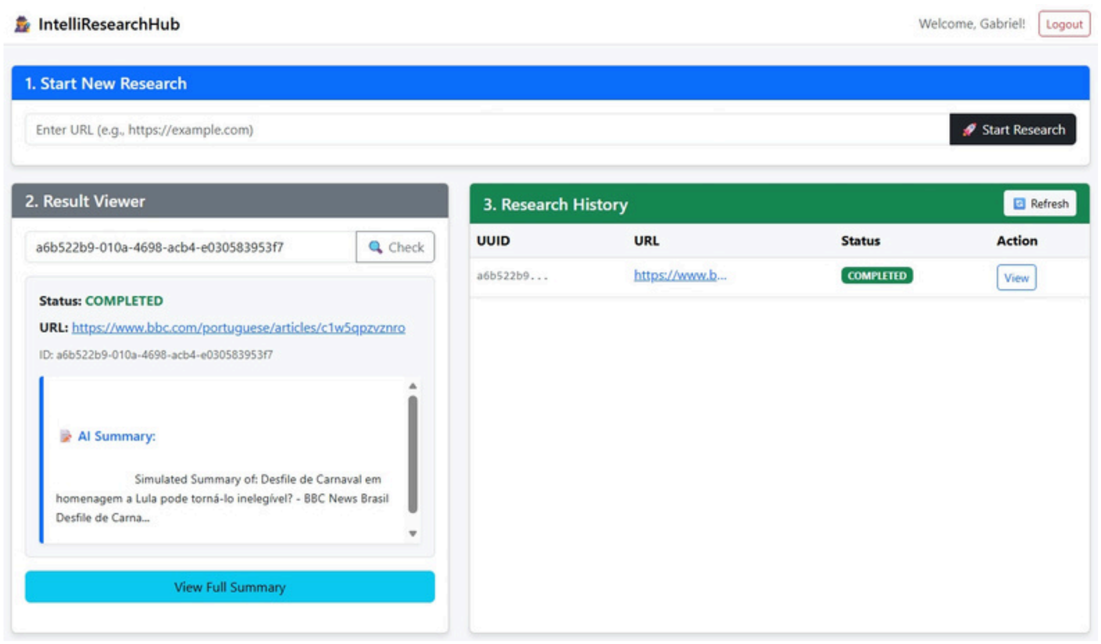
Enter URL (e.g., https://example.com) — Start Research

2. Result Viewer

Paste UUID here... — Check

3. Research History — Refresh

UUID	URL	Status	Action
a6b522b9...	https://www.b...	COMPLETED	View


IntelliResearchHub — Welcome, Gabriel! Logout

1. Start New Research

Enter URL (e.g., https://example.com) — Start Research

2. Result Viewer

a6b522b9-010a-4698-acb4-e030583953f7 — Check

Status: COMPLETED

URL: https://www.bbc.com/portuguese/articles/c1w5qpzvznro

ID: a6b522b9-010a-4698-acb4-e030583953f7

AI Summary:

Simulated Summary of: Desfile de Carnaval em homenagem a Lula pode torná-lo inelegível? - BBC News Brasil Desfile de Carna...

View Full Summary

3. Research History — Refresh

UUID	URL	Status	Action
a6b522b9...	https://www.b...	COMPLETED	View