ChatGPT

Using Semantic Knowledge Graphs for London Business School's Content Discovery

By Dinis Cruz and ChatGPT Deep Research

Introduction

London Business School's website hosts a vast amount of information – from program details and research insights to faculty profiles and news. Navigating this rich content and quickly finding relevant information can be challenging for users. This white paper proposes a solution based on **semantic knowledge graphs** to improve content discovery, personalization, and navigation on the LBS website. By leveraging advanced content extraction, knowledge graph construction, and large language model (LLM) analysis, we aim to create an intelligent layer on top of the existing site that helps both users and administrators interact with content more effectively.

In the sections that follow, we outline the overall approach and project plan. We describe how website content will be collected and transformed into a structured knowledge graph, and how various prototype user interfaces (UIs) can utilize this graph to deliver enhanced search and navigation experiences. We also discuss the integration of LLM-driven semantic analysis to annotate content with meaningful metadata (such as topics or sentiment), enabling dynamic filtered views and personalized content delivery. Finally, we consider administrative tools for curating the knowledge graph, and the potential for autonomous AI agents to continuously improve the system. The goal is to provide a clear blueprint – from data ingestion to front-end interfaces – for implementing a semantic graph platform that empowers LBS users to easily find and explore content of interest.

System Architecture Overview

At a high level, our solution will consist of three major components that work in tandem: (1) a **Content Extraction Pipeline**, (2) a **Semantic Knowledge Graph Database**, and (3) a suite of **User Interface Prototypes** for different use cases. Supporting these are existing tools and services developed in previous projects (such as *HTML processing* and *semantic text analysis* services) that will be adapted to the LBS context. Below is a brief overview of the architecture and key components:

- **Content Extraction Pipeline:** A backend process will crawl and retrieve pages from the LBS website, then convert each page's HTML into structured data. This involves parsing the HTML/DOM, extracting textual content, and normalizing the data. The pipeline will store raw and processed data in a dedicated repository. It uses a "Load-Extract-Transform-Save" approach – loading pages, extracting meaningful elements, transforming them into a standard JSON representation, and saving the results for further processing. This pipeline will be automated with Continuous Integration (CI) to keep the knowledge graph up-to-date as the website changes.

- **Semantic Knowledge Graph Database:** Using the parsed content, we will build a graph representation that captures relationships between different pieces of LBS content. For example, nodes in the graph might represent pages, sections, topics, or other domain-specific entities (like programs, people, research themes), while edges represent relationships (such as *"page A links to page B"*, *"professor X teaches program Y"*, or *"article Z is about topic Q"*). We plan to use a lightweight, serverless graph database (referred to here as **M-Graph DB**) to store and query this knowledge graph. The graph will be enriched with semantic metadata obtained via LLMs – for instance, linking content to taxonomy categories or tagging sentiment – to enable more intelligent queries and personalization.

- **User Interface Prototypes:** On top of the knowledge graph, we will develop various front-end interfaces to demonstrate how the semantic graph can enhance user experience. These UIs will range from simple text-based pages to rich interactive visualizations and personalized content views. The development will follow an *iterative flow development (IFD)* methodology: creating modular, atomic components (in pure HTML/CSS/JS or using Web Components) that can be easily adjusted or extended. This approach, combined with AI-assisted coding tools, will allow rapid experimentation with different layouts and interaction modes without breaking the overall system. We will start with basic prototypes (for debugging and proof-of-concept) and progressively move to more sophisticated interfaces, as detailed in the project phases below.

Underpinning these components are a few **existing tools and services** developed by our team, which we will reuse and adapt: an HTML parsing service that can convert raw webpages into structured JSON and extract textual snippets (with hashing for deduplication and change detection), a semantic text analysis service (powered by LLMs) that can classify or annotate text with attributes (like tone, topic, or taxonomy labels), and the CI/CD setup for automating data refresh and deployment. By leveraging these, we reduce development effort and focus on the LBS-specific knowledge graph and UI logic.

The following sections present the project plan in a phase-by-phase breakdown. Each phase introduces new capabilities on top of the previous ones, gradually evolving the system from basic data collection to an intelligent, interactive platform for content discovery.

Phase 1: Data Acquisition and Content Extraction

Objective: Gather the initial corpus of LBS website content and convert it into a structured format suitable for analysis.

In this first phase, we will build a crawler and extractor to retrieve web pages from the London Business School site and store their content in a structured way. Instead of attempting to ingest the entire site at once, we will start with a representative sample (for example, the homepage and a selection of top-level pages) to develop and test the pipeline. Key tasks in this phase include:

- **HTML Fetching:** Using standard HTTP requests, the system will download the HTML of selected LBS webpages. We may start with ~10 pages (e.g. the homepage and pages linked from it) as an initial dataset. The fetcher will operate like a proxy or crawler, but initially we can manually specify important URLs to include. Each fetched page's raw HTML will be saved to a repository (the *Content Repo*). Storing raw HTML ensures we have an exact snapshot of the page content as a baseline for parsing and for change tracking.

- **HTML to JSON Conversion:** We will process each saved HTML page through our existing **HTML service**, which parses the HTML into a structured JSON or dictionary representation of the DOM. This conversion will produce a machine-friendly version of the page, capturing the hierarchical structure of elements (divs, headings, links, paragraphs, etc.), attributes, and textual content. All textual nodes in the DOM will be extracted and replaced with unique identifiers (such as hashes).

For example, a paragraph's text might be replaced with a hash code, and the actual text content stored in a separate mapping. This yields two important artifacts per page: (1) a JSON file representing the page structure (DOM tree with placeholders), and (2) a text content file mapping each placeholder hash to the original text string. This separation allows us to efficiently track content changes and avoid storing duplicate text.

- **Handling Next.js Data:** The LBS website is built with Next.js, which often includes a global `__NEXT_DATA__` JSON script in each page. This JSON contains structured data (props, page state, etc.) that often replicates or complements the visible HTML content. We will parse the Next.js data object for each page to extract any useful information (such as page metadata, content blocks, or dynamic data not directly in the static HTML). Like the HTML, we will store this Next.js JSON data in a structured file and also extract textual content from it into hash-mapped form. Incorporating this ensures we don't miss content that is rendered or present only in the client-side data.

- **Initial Data Validation:** After extraction, we will have for each page a set of files (raw HTML, structured DOM JSON, text content hashes, Next.js data JSON). We will perform basic validation – e.g., ensuring that the text extracted from HTML plus Next.js covers the important visible content of the page, and that no major sections are missed. At this stage we may manually inspect or use diff tools to verify that the structured version faithfully represents the original page content.

By the end of Phase 1, we will have a small repository of LBS site content in a structured, machine-readable format. This repository acts as the foundation for building the semantic graph. Crucially, this process is repeatable and can be scaled up: once the method is verified on a handful of pages, we can expand to additional pages or sections of the site incrementally. The raw and processed data saved in this phase will be under version control (Git), which enables tracking changes to content over time and rolling back if needed.

Phase 2: Content Parsing and Domain Modeling

Objective: Refine the structured content and identify recurring patterns or components in LBS pages, in preparation for graph construction.

With the basic page data in hand, Phase 2 dives deeper into organizing and normalizing the content. The London Business School site likely has consistent design elements and content structures across its pages (for example, navigation bars, footer sections, content blocks for programs or research articles, etc.). This phase aims to recognize those patterns and begin forming **domain-specific content models** that will later translate into graph entities. Key aspects of Phase 2 include:

- **Domain Object Identification:** We will analyze the JSON page structures from Phase 1 to find common elements. For instance, many pages might share a header menu, footer, or sidebar – those can be abstracted as reusable components. More importantly, we look for conceptual components: e.g., a "Course Overview" section, "Faculty Profile" block, "Event listing", "News article snippet", and so on, depending on what content is present in the pages crawled. By comparing the structure and content of multiple pages, the system (with some guidance) can cluster elements that appear similar. In practice, this might involve writing rules or using LLM assistance to label sections of a page. For example, if a page has a title, an image, and a description paragraph consistently, we might label that pattern as a "Page Hero section". If multiple pages show a list of items with titles and dates, that could be identified as a "Listing component" (which could represent news listings, event listings, etc., depending on context).

- **Content Hash Mapping:** Using the text hash mapping from Phase 1, we will create a consolidated index of all unique text snippets across the pages. This helps in two ways: (1) detecting duplicate content (for instance, the same footer text or repeated descriptions) so we can store it once, and (2) tracking changes over time, as any content update will produce a new hash. During this phase, we'll confirm that each textual piece is captured and uniquely identified. The presence of duplicate hashes on different pages is a clue that those pages share a component or piece of information (like a common call-to-action or the school's address, etc.). We treat those as single content nodes that multiple pages reference, rather than separate items.

- **Normalization of Structures:** We will refine the JSON representations by cleaning or abstracting any page-specific noise. For example, dynamic IDs, tracking scripts, or timestamps that differ on every fetch can be removed or generalized so they don't impede change detection. The Next.js data will also be distilled to the parts relevant to content. This process results in a cleaner dataset where each page is broken down into a hierarchy of meaningful sections and content pieces.

- **Preliminary Ontologies (Draft):** At this stage, we begin thinking about ontologies – controlled vocabularies or category systems – that apply to LBS content. Without yet invoking heavy LLM analysis, we can start simple by using the site's navigation structure or headings. For example, top-level menu items like "Programs", "Faculty & Research", "About Us", etc., can serve as initial high-level categories in a taxonomy. We map pages or sections to these where obvious. The goal is to scaffold some basic classification of content which will later be refined. These draft ontologies will eventually be confirmed or expanded using AI, but having an initial framework is useful now to anchor the graph structure.

By the end of Phase 2, we will have transformed raw page data into a more **conceptual model** of the website content. This includes recognizing repeated components, consolidating identical content, and establishing initial categories. We effectively move from treating each page as an isolated document to understanding pages as collections of interrelated content objects. This lays the groundwork for the next phase, where we formally create the knowledge graph linking those objects.

Phase 3: Knowledge Graph Construction

Objective: Represent the website's content and relationships as a graph data structure, capturing how pages and content elements connect semantically.

With structured content and identified domain objects, we can now construct the **semantic knowledge graph** for the LBS site. In this graph, each node will represent an entity (page, section, content block, or even a concept/topic), and edges will represent relationships (structural or semantic links between entities). This phase involves designing the schema of the graph and populating it with the data from Phase 2. Key activities in Phase 3:

- **Graph Schema Design:** We will define what types of nodes and relationships make sense for this project. Likely node types include: *Page*, *Section/Component*, *Content Item (Text/Image)*, and *Topic/Category*. Relationship types might include: *"contains"* (Page contains Section, Section contains Content), *"links to"* (Page A links to Page B via a hyperlink), *"is of type"* (Section is of type e.g. "Faculty Profile"), *"has topic"* (Content Item is about Topic X), etc. This schema is flexible and will evolve, but starting with a clear model helps maintain consistency. For example, we might decide that each webpage is a node that has child nodes for each major section identified. Those sections then connect to text nodes or media nodes that hold the actual content. Additionally, pages can have connections to other pages (through hyperlinks or logical site structure).

- **Populating the Graph Database:** Using the schema, we will insert all extracted content into the **graph database (M-Graph DB)**. For each page processed, we create a *Page* node. We then create child nodes for significant sections or components on that page (as identified in Phase 2). Each content snippet (text block) becomes a node or an attribute on a node; if we choose to treat atomic text as separate nodes (especially if they are reused across pages), they will be connected via *"appears in"* or *"contained in"* relationships to their parent sections/pages. All hyperlink connections found in the HTML are added as edges between the corresponding page nodes (e.g., an edge from Page A to Page B labeled "links_to"). The initial taxonomy or category information we drafted can also be incorporated: for instance, if a page was classified under "Programs", we create an edge from that page node to a *Topic/Category* node labeled "Programs".

- **Serverless Graph Setup:** We will configure the graph database in a serverless or local environment suitable for the project. The M-Graph DB may be a lightweight in-browser graph (for client-side use) or a small hosted database. The emphasis is on ease of querying and visualization rather than massive scale (initially we are dealing with tens to hundreds of pages, which is manageable). The data from the Content Repo (JSON files, etc.) can be ingested via scripts to build the graph. Because our data is also version-controlled, re-populating the graph can be automated whenever new content is fetched or changes are detected.

- **Testing and Querying:** Once the graph is built, we will run some sample queries to ensure it reflects the site correctly. For example, we can query for all pages that link to a certain page, or retrieve the content of a specific section across pages. We can also test traversals like "find all content pieces that are categorized under Research and appear on a Faculty page", etc. Early queries verify that relationships (edges) have been created properly and that no content got lost in translation. If any issues are found (e.g., missing links or wrong hierarchy), we will adjust the schema or extraction accordingly.

At the end of Phase 3, we will have a foundational **knowledge graph of the LBS website**. This graph encapsulates both the structure of the site (how pages and sections are organized) and the interconnections (hyperlinks, shared content, category groupings). It is important to note that at this point, the graph is largely based on the explicit information from the site itself (the structure and links that were present). In subsequent phases, we will enrich this graph with additional semantic relationships inferred via AI, but even in its initial form, the graph is a powerful representation that can drive various visualization and navigation tools.

Phase 4: Continuous Integration for Ongoing Content Updates

Objective: Automate the data extraction and graph update process so that the knowledge graph stays current with the live website, without manual intervention.

Websites are living systems – content on the LBS site will change over time (new pages added, updates to existing information, etc.). To keep our semantic graph in sync with the site, we will establish a Continuous Integration/Continuous Deployment (CI/CD) pipeline in this phase. The pipeline will periodically run the extraction and graph-building steps (Phases 1–3) and update the repositories and database accordingly. Key components of Phase 4 include:

- **Automated Crawling and Refresh:** We will set up a CI job (for example, a GitHub Actions workflow or similar) that triggers on a schedule (e.g., nightly or weekly) or via manual trigger to perform the following: fetch a set of target pages from the LBS website, parse and process them into JSON and text as described, and commit any changes to the Content Repository. The set of pages to fetch can be configured – for instance, starting with a breadth-first crawl from the homepage up to a certain depth, or a curated list of important pages. Over time, we can expand the coverage. Initially, the pipeline might handle 10–20 pages; later it could scale to hundreds or the entire site, depending on performance and needs.

- **Change Detection and Version Control:** The pipeline will leverage Git version control to track changes in page content. When the extraction runs, it will produce updated JSON and text files. The CI process will compare these against the previous run's files. If nothing significant changed (e.g., only non-content differences like timestamps or random IDs), the pipeline can be configured to ignore or minimize those changes. If content has changed or new pages have been added, the pipeline will commit the updated files. Each commit thus represents a snapshot in time of the site's content. By examining the Git history, one can identify when content was added or modified. This built-in version history is valuable for both debugging and for any future feature that might show content evolution.

- **Content Repository Structure:** We will organize the content repository in a logical way – for example, one directory per page, containing that page's HTML, JSON structure, text hash file, etc. Alternatively, separate directories for raw HTML vs processed data can be used. The structure should make it easy for the UI or graph-building code to find the latest version of each page's data. The CI pipeline will ensure that after each run, the repository's main branch reflects the latest known state of the website content.

- **Graph Update Automation:** In addition to updating the content files, the pipeline will also update the knowledge graph (Phase 3) data. This could mean re-running a script to regenerate the graph from scratch using the new content files, or doing a more incremental update (adding/updating only the changed nodes). Given the moderate size of the data, a simple approach is to rebuild the graph fully on each run. This ensures consistency and can simplify handling deletions (if a page is removed from the site, a full rebuild will naturally drop it, whereas incremental update would need logic to detect and remove nodes). The end of the pipeline can include steps to deploy or make the updated graph accessible to the front-end (for example, exporting to a JSON graph file, or updating a cloud-hosted database).

- **Integration Between Repos:** We have three repositories in play – (1) the Code Repository (holding the pipeline code, parsing logic, etc.), (2) the Content Repository (holding the output data from the pipeline), and (3) the UI Repository (which we will discuss in the next phases for the front-end prototypes). The CI pipeline primarily operates in the context of the Code Repo but will push changes to the Content Repo. The UI Repo will include the Content Repo as a submodule or will fetch data from it as needed. In this phase, we'll configure this integration so that the UI layer always has access to the latest content and graph data published by the pipeline. For example, the UI build process might pull the Content Repo's files into its own build directory or reference them directly if using a client-side approach to load data.

By completing Phase 4, we establish an **automated, repeatable process** for keeping the semantic knowledge graph up-to-date. This means our subsequent work on user interfaces and semantic enhancements will always operate on current data. It also means the system can be left running and continue to function as the LBS website evolves, without requiring constant manual updates. This robustness is important for any real-world deployment at LBS, as it ensures the tools we build remain accurate and useful over time.

Phase 5: User Interface Prototypes for Content Exploration

Objective: Develop a series of front-end interfaces to demonstrate and test how the structured content and graph can be used to improve content discovery.

With the data and backend in place, we turn to creating the user-facing side of the project. In Phase 5, we will build multiple prototype UIs, each exploring different ways of presenting the LBS content through the lens of our semantic graph. These prototypes will range from very basic text renderings (useful for debugging and as LLM-friendly views) to more complex interactive sites. We plan to use a method of incremental, modular development (the Iterative Flow Development approach), which means building the UI in small pieces that can be individually tested and adjusted. Each prototype mode will reside in the UI repository (likely as separate pages or sections of the app). The prototypes include:

5a. Text-Only Content View

The first UI mode is a **plain text or Markdown-based view** of the extracted content. This prototype will display the textual content of pages without the original site's styling or layout – essentially a simplified, content-centric version of each page. The purpose of this mode is to verify that we have captured all key content and to provide a clean reading interface that's also friendly for large language models or further analysis. Key features:

- Each page's content is presented in a linear text format (optionally with basic Markdown formatting for headings, lists, etc., derived from the structure). Non-essential navigation or decorative elements might be omitted to focus on primary text.
- Content that was originally separated (e.g., accordions, tabs) might simply be shown in sequence, since our interest is in raw content exposure.
- This view helps ensure no content is hidden or lost in the extraction process. It can also serve as a baseline for comparing how the same content could be rendered differently in other modes.
- From an implementation perspective, this is straightforward: the UI can load the text content JSON (hash-to-text mappings) for a page and print out each piece in order, perhaps preceded by a label or placeholder showing where it came in the structure.

The text-only view, while not intended for end-users in final form, is a valuable development tool. It confirms the integrity of data extraction and can even be used as input for LLMs (since it strips away boilerplate and focuses on core text). It's also extremely fast and lightweight, making it a good sanity-check page.

5b. Interactive Content Visualization Dashboard

The second prototype explores **visual representations of the site's structure and content data**. Instead of focusing on page-by-page text, this mode will provide an overview through graphs, charts, or other visualizations. The goal is to illustrate relationships and patterns in the content that might not be obvious from looking at a single page. Possible features of this dashboard include:

- **Site Map Graph:** A diagram of the website's structure as captured in our knowledge graph. For example, nodes could represent pages (with their titles), and arrows could represent links between pages. This could be visualized using a library like D3.js or a simpler SVG approach. Users (or developers in this context) could see clusters of pages and how they interconnect. This helps identify key navigation hubs or isolated pages.

- **Component Reuse Visualization:** Charts showing which content components appear on how many pages. For instance, a bar chart for “Top 10 reused text snippets” (which might highlight common footer text or repeated calls-to-action), or a graph linking a content snippet node to all pages that include it. This can demonstrate the degree of content reuse and consistency across the site.
- **Content Statistics:** Display some aggregate statistics derived from the content repository – such as number of pages processed, total word count, average words per page, distribution of content by category, etc. This information might be shown in tables or simple graphs (pie charts, histograms).
- **Mermaid.js Diagrams or Other Formats:** We can leverage Markdown-friendly diagram tools (like Mermaid) to render some relationship diagrams purely on the client side, if appropriate. For example, a Mermaid flowchart could depict a hierarchy (like the site navigation tree).

This visualization dashboard serves multiple purposes. It is a **debugging and analysis tool** for the team, to ensure our graph data makes sense, and it’s also an impressive demonstration to stakeholders of what a semantic approach can reveal. For instance, LBS staff might be interested to see a graph of how different sections of the site connect, or identify content that is orphaned (not linked from anywhere). The interactive nature (hovering on nodes to see details, clicking to focus on a subgraph, etc.) can make it engaging. Implementing this will likely involve integrating some JS libraries for graphs/charts, but we will keep it lightweight and static (all data can be loaded from the content JSON files or a precomputed graph JSON).

5c. Reconstructed Static Website (Minimal & Styled Versions)

Next, we will create a prototype that uses our extracted content to **reconstruct a portion of the LBS website** as a static site. This means generating web pages that contain the same information as the original LBS pages we processed, but built independently (and without any of the original site’s underlying code). There are a few iterations we can attempt here:

- **Basic Static Version:** We start with a minimal HTML version of each page using our content. This would involve taking the structured JSON for a page and rendering it into an HTML page that resembles a very simple website. For example, all the main sections of the page could be laid out in order, with basic HTML markup (headings, paragraphs, lists, images, links, etc.), but no fancy styling. The navigation menu might be simplified or omitted initially. The idea is to prove that using only our stored content, we can serve a human-readable page that contains everything the original did. Links on these pages would point to other reconstructed pages (only those that we have content for). If a link’s target page wasn’t yet scraped, we may disable that link or just show it as plain text to avoid broken links. Essentially, this static site is self-contained and limited to the subset of pages we have processed.

- **Enhanced Styling Version:** Once the basic version works, we can incrementally add CSS styles or simple design elements to improve readability and appearance. We might introduce a consistent LBS-inspired style (without copying proprietary assets, but we can mimic the color scheme or layout to some degree). We could also include a simplified navigation bar and footer on each page to make it feel more like a coherent site. This version could have light/dark themes or different color palettes, just to show how flexible the content is when we separate it from the original site’s styling. Each style variation can be an alternative stylesheet or a toggle, demonstrating that we can skin the content in multiple ways easily.

- **Original Layout Approximation:** As a more advanced challenge, we can try to **replicate the original site's look and layout** using our content but built with modern, clean code. This would involve analyzing the original CSS/JS to some extent for layout ideas, but then implementing our own version. The aim is to create a page that, to a casual observer, looks almost like the real LBS page, yet is actually running purely off our data (and importantly, without the heavy scripts, trackers, or complex frameworks – a more efficient clone). We can utilize Web Components or modern CSS features to implement common design patterns found on the site (like grid layouts, responsive menus, etc.). Because we only focus on the content we have (maybe a handful of pages), we can manually fine-tune the layout for those pages. This serves as a proof-of-concept that our approach could deliver an experience very close to the official site, but potentially faster and more user-focused (since we control exactly what loads and can optimize it).

In building the reconstructed site, we ensure that **only the content we've extracted is used** – no live calls to the original site, and no dependence on its infrastructure. This means the prototype site is completely static (it could even be hosted on GitHub Pages or similar), and it demonstrates the portability of the content. One can imagine an scenario where, if the knowledge graph content were comprehensive, LBS could have an alternative or backup site generated from it, or sub-sites tailored to specific audiences. Phase 5c's prototypes lay the foundation for that idea.

These UI prototypes in Phase 5 not only validate that our data can be used to render real user-facing pages, but they also start to hint at the possibilities once we have full control over presentation. In subsequent phases, we will extend this further by introducing semantic and personalized reorganisation of content on these pages.

Phase 6: Semantic Enrichment with LLM Analysis

Objective: Augment the knowledge graph and content repository with additional semantic metadata (such as sentiment, topics, or keywords) using Large Language Models, and leverage this data for new ways of filtering or viewing content.

Up to Phase 5, our system deals mostly with explicit information – the content as it is on the site and straightforward relationships (structure, links). Phase 6 introduces **AI-driven insights** into the mix. By employing an LLM (via our semantic text service), we can extract deeper attributes from the content that are not directly given on the site. These might include the emotional tone of content, the topics or themes discussed, the intended audience, etc. We then incorporate these attributes into the graph or as annotations in the content JSON, enabling advanced query and UI features. The steps in this phase include:

- **Sentiment and Tone Analysis:** For each piece of text (or each page as a whole), we can use an LLM to evaluate the sentiment (positive, neutral, negative) or overall tone (e.g., formal, enthusiastic, academic, promotional). For example, a news article about a successful event might be tagged as having a positive/upbeat tone. We will generate a **"positivity score"** or a simple tag for relevant content blocks. This data can be stored in a new JSON file mapping text hashes to sentiment values, or directly added as attributes on nodes in the knowledge graph (e.g., a property on a Content node).

- **Topic Tagging and Categorization:** Using either LLMs or domain-specific classifiers, we will assign topics or keywords to content. This goes beyond the basic taxonomy from Phase 2. For instance, an LLM could read a piece of text and identify that it's about "entrepreneurship" or "finance" or "organizational behavior" (topics relevant to a business school). We can compile a list of key themes the LLM should look for (perhaps aligning with LBS's departments or research areas), or let it free-form tag and then cluster the results. The outcome will be that each page or section has a set of associated topics. These topics themselves become nodes in the graph (if not already) with *"has_topic"* relationships connecting them to content nodes.

- **Audience/Perspective Annotations:** Another semantic angle is identifying for whom a piece of content might be most relevant – e.g., is a page aimed at prospective students, current students, alumni, faculty, or researchers? Often the site structure implies this, but an LLM can help confirm or classify ambiguous cases. We can label pages or content with one or more audience tags. This will be useful later when we create persona-based views (Phase 8).

- **Data Integration:** All these annotations (sentiment, topics, audience, etc.) need to be integrated back into our system. Practically, we will create new data files or graph entries as needed. For example, we might produce a JSON file `analysis/sentiment.json` that lists content hash -> sentiment score, and `analysis/topics.json` listing content hash -> [topic tags]. The knowledge graph is then updated to include edges from content nodes to new *Sentiment* or *Topic* nodes (or simply store them as properties if the graph DB allows property storage). The advantage of using the graph here is that it naturally connects these new semantic nodes to the existing structure, which means queries can combine structural and semantic aspects (e.g., "find positive-tone articles about entrepreneurship in the Faculty & Research section").

- **Quality Check and Refinement:** We will verify a sample of the LLM-generated annotations for accuracy and relevance. LLMs might occasionally produce incorrect or overly broad tags, so a validation step is wise. If needed, we can refine the prompts given to the LLM or set up rules to ignore irrelevant tags. This step ensures the semantic metadata we attach is trustworthy enough to drive user-facing features.

With the content now semantically enriched, we unlock the ability to view and filter content in ways not possible on the original site. For example, a user could ask to see only the news articles that have a highly positive tone, or to list programs that mention "data science" as a topic. These capabilities will be explored in the UI in the next phase. Phase 6 essentially transforms our knowledge graph from a structural map of the site into a **semantic network** where nodes carry meanings and context that align with how users might think about the content.

Phase 7: Knowledge Graph-Driven User Interfaces

Objective: Create advanced user interfaces that leverage the full semantic knowledge graph – including ontologies and LLM-generated metadata – to provide intuitive navigation, search, and discovery of content across the site.

Having built a rich knowledge graph (with structural and semantic relationships), Phase 7 focuses on using that graph to power user experiences that go beyond the original site's capabilities. We will develop interfaces that allow users to explore content by topic, by relationship, and by custom criteria. This is where the true value of the semantic approach becomes visible to end-users. We outline several interfaces and features below:

7a. Ontology-Based Content Navigation (Per-Page and Site-Wide)

One of the key outputs of our semantic analysis is an **ontology/taxonomy** of the site's content. In Phase 6, we tagged pages and sections with topics and possibly identified hierarchical categories. In this sub-phase, we will present those ontologies in the UI as a means of navigation:

- **Page-Level Outline with Semantic Tags:** For each page, we can generate an enhanced outline or sidebar that shows the structure of that page along with tags indicating the topics or categories of each section. For instance, on a page about a Masters program, the sidebar might list sections like *Overview*, *Curriculum*, *Admissions*, etc., with each section tagged by themes such as "Academics", "Student Life", or "Requirements" as appropriate. This gives a user a quick semantic summary of what the page contains. It's like a table of contents, but enriched with context. The user can click on a tag to see related content (e.g., clicking "Admissions" could suggest other program pages or articles related to admissions, via the graph).

- **Site-Wide Topic Browsing:** We will create a UI that lists all the major topics or categories extracted across the site (this essentially reflects the site-wide ontology). Users can click on a topic to see all content items (pages, or even specific sections) related to that topic. This is an alternative entry point to the content, different from the traditional site menu. For example, a topic like "Entrepreneurship" might aggregate links to faculty profiles, research articles, courses, and news stories all in one place, because our knowledge graph knows those items are semantically linked. This breaking down of silos is a powerful feature of the knowledge graph approach.

- **Ontology Graph Visualization:** In addition to a list or menu-based browsing, we could incorporate a visual graph view for the site-wide ontology. This would show high-level categories connected to sub-categories or directly to content nodes. Users (or content managers) can visually explore how broad themes branch into more specific ones and where content sits on that graph. For instance, a "Faculty & Research" node might connect to subtopics like "Finance", "Marketing", "Strategy", etc., each of which links to related content pages. This visualization could be interactive, allowing zooming in on a portion of the ontology and clicking through to content.

7b. Filtered Search and Dynamic Views

Using the metadata from Phase 6, we can implement advanced filtering and search capabilities in the UI. This goes beyond a simple keyword search box by allowing users to refine what kind of content they want to see using semantic filters:

- **Sentiment Filter:** Imagine a user (or LBS content manager) wants to find only upbeat stories to feature in a newsletter. We could have a slider or filter for sentiment/tone. For example, the user could set "Tone: Positive" and see a list of news articles or testimonials that the LLM flagged as positive in sentiment. Conversely, an analyst might look for negative sentiment mentions (perhaps to spot any critical feedback on the site). The UI would query the graph for content nodes that meet the sentiment criteria. This kind of filter is novel, as it's not typically available in standard site search.

- **Audience/Persona Filter:** If we tagged content by intended audience, the UI can let a user filter content for "Prospective Student", "Current Student", "Alumni", "Faculty", etc. Selecting one of these personas would reconfigure the view to show only the pages or sections relevant to that audience. For example, an "Alumni" filter might surface alumni success stories, alumni events, donation pages, etc., across the whole site, even if those are normally in different sections. This feature effectively personalizes the site on the fly for different user groups.

- **Combined Queries:** We will enable combining multiple filters or query parameters. A user could, for instance, search for content that is **category = “Research”** and **topic = “Data Science”** and **tone = “Neutral/Academic”**. The UI would return results (pages, or maybe specific sections within pages) that match all those criteria. This demonstrates the power of the graph – since all these attributes are connected to content nodes, the query engine can intersect them to find the overlapping set. From a user perspective, this is a very targeted search that traditional navigation would not easily allow.

- **Presentation of Results:** For any filtered view or search result, we will design a clean results page. Results might be listed as cards or a list of links with some context snippet. We can highlight why a result was shown (e.g., tags or categories that matched the filter). Because our system knows the structure, we could even scroll the user directly to the relevant section of a page if the query was at the section-level granularity. For example, if a user filtered by “Admissions” and “MBA program”, we might take them directly to the Admissions section of the MBA Program page, rather than just the top of that page.

7c. Knowledge Graph Exploration Interface

This sub-phase involves a more free-form **graph explorer tool** for power users or developers. It would allow direct interaction with the underlying knowledge graph:

- **Graph Query Console:** Provide a simple interface where one can input graph queries (perhaps in a query language or a simplified syntax we define) and get results. This is more of an admin or developer feature, but it helps to have a way to run arbitrary queries, e.g., “Show me all pages that have more than 5 outgoing links and belong to category X”, or “List all unique content snippets that appear on at least 3 different pages”. If we use a standard like SPARQL or Gremlin for the graph, this console could accept those queries. Otherwise, even a custom dropdown-based query builder could be provided for common query patterns.

- **Interactive Graph GUI:** Beyond the query console, we can have a visual graph exploration mode. The user could start from a particular node (say, a page or topic) and see its immediate connections, then expand nodes on the fly. This is akin to mind-map or knowledge graph viewers, where you can wander the graph by clicking. For LBS, one could start at “London Business School” node, see connections to main sections (Programs, Faculty, etc.), click on “Faculty”, see connections to individual faculty pages or research topics, and so on. This mode would be excellent for demonstrating the richness of connections in the content, and might even uncover relationships that were not obvious (like a research topic connecting faculty and news articles and course descriptions together).

- **Debugging and Improvement Insight:** The graph exploration interface also serves as a diagnostic tool. If something is mis-categorized or isolated in the graph, it will show up visually. For example, if a page isn’t linked to any topic, it might float alone – indicating we should perhaps add a link or check why it wasn’t tagged. Similarly, if a topic node has an overwhelming number of connections, maybe it’s too broad and we might want to subdivide it. This direct feedback from the interface can guide further refinement of the semantic annotations.

By the end of Phase 7, we will have demonstrated how the knowledge graph can directly fuel an enhanced user experience. From guided navigation by topic to powerful search and even raw graph exploration, these interfaces showcase a level of interaction that traditional websites (with static menus and keyword search) cannot easily provide. For London Business School, this could mean users find information faster and in more intuitive ways – for example, a prospective student interested in entrepreneurship can traverse from courses to faculty to research to events on that theme seamlessly, without needing to know the site's structure. This phase solidifies the argument for semantic approaches by making the benefits tangible.

Phase 8: Personalized and Contextual User Experiences

Objective: Utilize the semantic graph and content metadata to dynamically tailor content presentation for different user personas and contexts, demonstrating personalized experiences.

Building on the filtering and ontology work, Phase 8 focuses on **personalization** – the ability to present content that is most relevant to a specific type of user or a specific context. London Business School serves multiple audiences (prospective students, current students, alumni, faculty, researchers, corporate partners, etc.), and each audience has different content priorities. With our semantic graph, we can create custom views or even entire site variants optimized for each persona. Key initiatives in this phase include:

- **Persona-Specific Portals:** We can create dedicated entry pages or dashboards for each major audience. For example, a "Prospective Students Portal" page that aggregates content from across the site relevant to prospects: upcoming admissions events, application guidelines, program overviews, student testimonials, etc. Because our graph knows which content pieces are tagged for prospective students and their topics, we can pull them together even if originally they live in separate sections of the official site. Similarly, an "Alumni Portal" might show alumni news, opportunities to engage, continuing education programs, and donation information in one place. These portals effectively reorganize the site's content semantically rather than following the site's actual menu structure.

- **Dynamic Page Customization:** We can also personalize at the level of individual page views. For instance, suppose a faculty profile page is being viewed by a corporate partner interested in research collaboration – we might emphasize that faculty's industry projects or consultancy work on the page. Or if the same profile is viewed by a prospective PhD student, we might highlight their publications and courses taught. Technically, this could be achieved by having the UI detect a selected persona (perhaps via a toggle or by login if that were in scope) and then using the graph to rearrange or filter sections of the page. While full dynamic personalization might be beyond a prototype's scope, we can simulate it by offering a dropdown in the UI to "View this page as: [Prospective Student / Current Student / Alumni / etc.]". The content would then show/hide or reorder based on what that persona is likely looking for.

- **Content Recommendations:** Personalization can also mean recommending related content that aligns with the user's interests. Using the semantic graph, whenever a user is viewing something, we can suggest *"You might also be interested in..."* with links that share topics or other attributes. For example, on a page about a finance program, recommend finance-related research articles or news. On a faculty page, recommend their latest research or similar faculty in the same department. These recommendations are powered by the *"topic"* and *"ontology"* connections in the graph – essentially doing a neighborhood search around the content being viewed. This adds a layer of content discovery that the current site might not have, and keeps users engaged by showing them relevant material without requiring a manual search.

- **Multi-Language or Localization Considerations:** While the current scope is primarily within the English content of LBS, the semantic approach could facilitate multi-language support in the future. If LBS ever provides content in other languages, our system could potentially align equivalent content across languages via the graph (for instance, linking an English page to its translated counterpart as the same concept node). Even without actual translated content, we could demonstrate the framework's readiness by perhaps machine-translating a page as a proof of concept and linking it. This shows that personalization can extend to language preference as well – a user could flip to another language view if available.

- **Performance and Experience:** We will ensure that these personalized or dynamic views remain performant. Since the heavy lifting is done by the graph queries and the data is all local (no need to call external services for each recommendation or filter), the UI can update quickly. Caching mechanisms can be used for expensive queries. We will also pay attention to UX – making sure that when filters are applied or persona toggles are switched, it's clear to the user what changed and why those pieces of content are being shown. The interface should remain clean and not overwhelm the user with too many controls at once; a guided experience (maybe a wizard for new prospective students, for example) could be another angle.

By implementing Phase 8, we demonstrate the **practical personalization potential** of the semantic knowledge graph. For LBS stakeholders, this can illustrate how the same underlying content can be repackaged in multiple ways to serve different needs without duplicating effort – the graph acts as a single source of truth that feeds various front-ends. It can increase user satisfaction by delivering relevant content faster, and potentially improve content management efficiency (since updates in one place automatically flow to all personalized views that include that content).

Phase 9: Editor and Admin Feedback Loops

Objective: Provide tools and interfaces for LBS content editors or administrators to review, refine, and influence the semantic graph and its outputs, ensuring the system remains accurate and aligned with human expectations.

While our system heavily leverages automation and AI to build the semantic structure, human oversight and input are invaluable. Phase 9 introduces an administrative interface where LBS staff (or the project team) can interact with the knowledge graph and annotation rules to correct or enhance them. The idea is to create a **feedback loop**: the AI proposes semantic relationships, humans review and adjust, and the AI/system incorporates those adjustments. Key features and tasks in this phase:

- **Graph Curation Dashboard:** An interface for content editors to see the ontologies, tags, and relationships that the system has generated. This could be a specialized view of the graph explorer from Phase 7c, focused on curation tasks. For example, it might list all topics identified along with the pages under each. Editors can quickly scan if something was mis-tagged (e.g., a page that doesn't actually relate to a topic appears under it) and then correct it. Correction could mean removing a tag from a page, or moving a page to a different category. The UI would allow such edits in a user-friendly way (e.g., checkboxes, drag-and-drop between categories, or an "edit tags" dialog for a page).

- **Override and Annotation Tools:** In cases where the automated system didn't catch a nuance, editors should be able to add their own annotations. For instance, if an important new topic emerges (say LBS launches a new initiative not recognized by the AI), an editor can create a new node for that topic and link relevant content to it manually. Or if a certain piece of text should be treated in a special way (maybe a tagline that should always be associated with a brand tag), they could annotate that. These human-provided inputs will be stored so that they persist through pipeline runs (meaning the pipeline should merge human edits with AI results, not overwrite them – which may require careful design, such as storing manual overrides in a separate file or in the graph with a flag).

- **Quality Reports:** The admin interface could highlight potential issues for editors' attention. For example: content pieces that are unlinked (not found in any page, perhaps an orphan in the dataset), or topic nodes that have only one item (maybe noise tags that could be eliminated), or very large clusters (which might indicate a too-broad category). By surfacing these, editors know where to focus their curation efforts for maximum effect.

- **Versioning and Audit Trail:** Any changes made by humans via the curation UI should be recorded (possibly also via Git or a change log). This way, one can track what adjustments were made and why. This is important for transparency – if down the line someone wonders why a certain page is tagged a certain way, there will be a record if it was a manual decision. Moreover, it allows rollback if a mistake is made. We might integrate this with Git by having the admin UI commit changes to a "Manual Overrides" JSON file in the content repo, for instance.

- **Feedback to AI for Continuous Improvement:** Although it might be beyond the immediate scope, it's worth noting that the corrections made by editors could be fed back into the AI models to improve them. For example, we could retrain or re-prompt the LLM using examples of tags that humans changed, to refine its criteria. Even if we don't implement a full feedback learning loop now, we will structure the system in a way that such improvements are possible in the future (for instance, maintaining a dataset of false positives/negatives for topics and feeding that to model maintainers).

By empowering human experts to shape the knowledge graph, Phase 9 ensures that the system doesn't operate as a black box. LBS content managers retain control over the messaging and structure. The semantic graph becomes a collaborative effort: AI does the heavy lifting to propose structure at scale, and humans do the fine-tuning where their domain expertise is needed. This collaboration can lead to a highly accurate and institutionally-aligned knowledge graph. Moreover, the presence of an admin interface and process helps with **governance** – an important aspect if LBS were to rely on this system in production. There would be clarity on how things can be updated or fixed, and trust in the data would be higher.

Phase 10: Autonomous Agents and Future Enhancements

Objective: Leverage autonomous AI agents and advanced automation to further enhance the system, explore new ideas, and maintain the knowledge graph-based site with minimal human intervention.

In the final phase, we look toward the future of this semantic platform. With all the pieces in place (data pipeline, knowledge graph, enriched metadata, multiple UIs, and admin feedback), the system is ripe for taking advantage of **autonomous AI agents** to drive continuous improvement and creative experimentation. While this phase is forward-looking and exploratory, we outline a few ways such agents could be applied:

- **Automated Content Generation & Suggestions:** AI agents (such as advanced GPT-based systems) could analyze the content graph and identify gaps or opportunities. For example, an agent might notice that a particular topic is trending in the business world but the LBS site has little content on it – it could suggest creating a new article or page for that topic. Or it might find that certain pages are frequently accessed together and suggest a combined guide or a new navigation link to link them. These suggestions could be presented to LBS content teams for consideration, effectively acting as an AI content strategist.

- **Dynamic UI/UX Optimization:** Agents could also experiment with the user interface configurations. Using the modular UI components developed, an AI agent could perform A/B tests or heuristic evaluations on different layouts or content arrangements for achieving certain goals (e.g., maximizing user engagement or simplifying navigation). For example, an agent might hypothesize that showing personalized news on the homepage for returning users would improve engagement, and it could simulate or even live-test that with a subset of users. Over time, the agent could fine-tune the UI by learning from user interactions (click patterns on the knowledge graph explorer, etc.). Essentially, the site could continuously evolve its presentation using AI guidance.

- **Multi-Agent "Swarm" for Maintenance:** We might deploy a collection of specialized agents, each with a certain role in maintaining the system. One agent monitors the LBS site for structural changes (like a new section added) and ensures our crawler covers it. Another monitors the graph for anomalies or performance issues. Another agent could handle user queries in natural language by translating them into graph queries and returning answers (making the site search more like a chatbot interface). Using a framework like **Claude/Cloud Flow** or other agent orchestration systems, these agents can communicate and coordinate. For example, if a user asks a complex question ("Show me all finance-related research since 2020 with female faculty involvement"), one agent might break this down: a semantic parsing agent interprets the question, a data agent queries the graph, and a response agent formats the result.

- **Learning from User Behavior:** If the enhanced site is deployed to actual users, we will have new data on how users navigate the semantic features. Agents can analyze this behavior to identify improvements – perhaps users often switch to a dark theme, or frequently use the persona filter for "Prospective Student". Knowing this, the agent might prioritize those features (like making the prospective student view more prominent). If some topics are rarely explored, the agent could investigate why – maybe the labeling is unclear, etc., and suggest changes. This is essentially using AI to do UX research and site analytics on the fly.

- **Future Extensions (Out of Scope Now but Enabled):** We note that while not immediate, the architecture we built can support other advanced features. For instance, integration with voice assistants (an agent that can answer spoken questions about LBS by leveraging the knowledge graph), or integration of external data (like pulling in relevant social media or news content related to LBS topics into the graph). The modular design means such extensions can plug in without reworking the core.

Phase 10 is about setting the stage for the project's evolution into an intelligent, mostly self-sustaining ecosystem. By introducing autonomous agents into the workflow, we can achieve a sort of *continuous improvement loop*, where the system not only updates itself with new content but also learns to present and organize content in better ways over time. For London Business School, this means the knowledge platform could remain at the cutting edge, adapting to user needs and institutional changes dynamically. It’s an ambitious vision, but even incremental steps in this direction (like an AI-assisted content suggestion here or an automated UX tweak there) can compound to make the platform significantly smarter and more effective year over year.

Conclusion

In this white paper, we have outlined a comprehensive plan to revolutionize content discovery and user experience on the London Business School website through the use of semantic knowledge graphs and AI technologies. The approach starts from the ground up: extracting raw HTML content and systematically structuring it into a rich knowledge graph that mirrors the site’s information architecture and relationships. Building on that foundation, we integrate large language model-driven analyses to imbue the graph with semantic depth – identifying sentiments, topics, and contextual relevance that transform static webpages into a dynamic, queryable knowledge network.

We proposed and detailed a series of prototype user interfaces demonstrating how this semantic network can be harnessed: from simple text views and data visualizations for developers, to reconstructed site pages, to advanced topic-based navigation and personalized portals for various user personas. These prototypes show that by separating content from presentation and adding intelligence, we can reassemble and present information in user-centric ways that were not possible with the original site alone. Users can find what they need faster, discover related content effortlessly, and view information through lenses that matter to them (such as topic, sentiment, or role).

Crucially, our plan includes mechanisms for maintaining and improving this system over time. The CI pipeline ensures the knowledge graph updates alongside site content changes, while the admin feedback loop empowers LBS staff to guide the AI’s understanding and categorization, keeping the human in control of the narrative. Furthermore, looking ahead, the introduction of autonomous agents hints at a future where the platform could self-optimize and grow more insightful the more it is used, truly becoming a living part of the digital campus.

For London Business School, implementing this project would mean providing students, faculty, alumni, and other stakeholders with a cutting-edge tool to engage with the school’s knowledge base. It aligns with a modern vision of digital presence – one that is intelligent, adaptive, and deeply user-focused. The collaboration between Dinis Cruz and ChatGPT (Deep Research) on this proposal itself exemplifies the fusion of human expertise and AI assistance that underpins the project’s philosophy. By crediting the development of this semantic knowledge graph platform to both human and AI contributors, LBS can showcase innovation in not just the end product, but in the very process of creation.

In conclusion, the use of semantic graphs and AI for content management is a forward-looking strategy that can significantly enhance how information is organized and consumed. We are confident that the phased plan outlined here provides a clear pathway to implement this vision. With each phase delivering incremental value – from data insights to user-facing improvements – London Business School can iteratively build and refine a system that ultimately sets a benchmark for academic and institutional websites. We look forward to the opportunity to bring this project to life, creating a smarter web experience for the LBS community.
