Architecting the Aevenalytics Semantic Core: A Blueprint for Ontology-Driven, Multilingual Knowledge Graph Construction

Part I: The Foundational Layer - High-Fidelity Multilingual Representation

The efficacy of any advanced knowledge system is contingent upon the quality of its foundational representation layer. This layer is responsible for transforming unstructured and semi-structured text, irrespective of its original language, into a high-fidelity, numerical format that captures deep semantic meaning. All subsequent processes, including automated knowledge extraction, relationship inference, and cross-lingual analysis, are fundamentally dependent on the precision and richness of these initial representations. This section details the strategic selection of a state-of-the-art multilingual embedding model and provides a comprehensive blueprint for its high-performance implementation within the Aevenalytics C#/.NET ecosystem, establishing the technological bedrock for the entire knowledge graph architecture.

1.1. Selecting the Optimal Multilingual Embedding Model: intfloat/multilingual-e5-large

The choice of an embedding model is the single most critical architectural decision in this system. After a thorough review of available technologies, the intfloat/multilingual-e5-large model is recommended as the optimal choice. This selection is not based on generic benchmarks but on the model's specific design, training methodology, and proven capabilities in tasks that directly align with the goals of knowledge extraction and semantic linkage.

Justification and Provenance

The intfloat/multilingual-e5-large model is a highly specialized text embedding model

engineered for superior performance in retrieval and semantic similarity tasks.[1] Its architecture and training regimen distinguish it from general-purpose multilingual models, making it uniquely suited for the demands of this project.

The model's foundation is xlm-roberta-large, a powerful cross-lingual transformer that provides a robust initial understanding of syntax, grammar, and semantics across 100 different languages.[1] This strong starting point is then enhanced through a sophisticated, two-stage training process that imbues it with the specific capabilities required for high-precision knowledge work.

The first stage consists of a massive-scale contrastive pre-training phase. The model was trained on over one billion weakly supervised, multilingual text pairs sourced from diverse corpora, including filtered mC4, CC News, Wikipedia, and NLLB translation pairs.[1] In this stage, the model learns to map semantically similar text fragments—such as a document title and its corresponding content, a question and its answer, or a statement and its translation—to proximate locations within its high-dimensional embedding space. This process is language-agnostic, teaching the model the fundamental task of recognizing semantic equivalence across linguistic boundaries.

The second stage involves supervised fine-tuning on a curated collection of high-quality, human-labeled datasets. These datasets, which include well-regarded benchmarks like MS MARCO (English), DuReader Retrieval (Chinese), and MIRACL (16 languages), are specifically designed for information retrieval tasks.[1] This fine-tuning stage sharpens the model's ability to discern subtle differences in relevance, conditioning it to produce embeddings that are highly discriminative for semantic search. This is precisely the capability needed to identify the most salient keyphrases within a document, as the process is analogous to finding the phrases that are most semantically similar to the document's overall theme.

Model Specifications and Capabilities

The multilingual-e5-large model is a deep architecture, featuring 24 transformer layers that culminate in a 1024-dimensional embedding vector.[1] This high dimensionality provides the capacity to encode fine-grained semantic nuances, which is essential for distinguishing between closely related but distinct concepts. While its primary design is for retrieval, its rich semantic representations make it highly effective for a range of downstream NLP tasks, including text classification, sentiment analysis, and even as a feature extractor for translation systems.[2] This inherent versatility represents a strategic asset for Aevenalytics, providing a single, powerful model that can serve multiple future use cases beyond the initial knowledge graph construction.

Critical Usage Requirement

A critical implementation detail, stemming directly from the model's contrastive training

protocol, is the mandatory use of specific prefixes for all input texts. To achieve optimal performance, each text string fed to the model for embedding must be prepended with either "query: " or "passage: ".[1] This instruction is not merely a suggestion but a requirement to align the input with the format the model learned during its fine-tuning phase. For tasks other than retrieval, such as classification or clustering, the
"query: " prefix is the recommended default.[1] In the context of keyphrase extraction, where candidate phrases are evaluated for their similarity to the source document, both the document (acting as the passage) and the candidate phrases (acting as queries) should be prefixed accordingly to ensure they are projected into the correct, comparable regions of the embedding space.

1.2. Achieving High-Performance Inference with ONNX Runtime in C#

Deploying a large transformer model like multilingual-e5-large into a production C# environment requires a robust, high-performance inference engine. The Open Neural Network Exchange (ONNX) format and its associated ONNX Runtime provide the ideal solution, serving as a bridge between the Python-centric world of model development and the enterprise-grade.NET ecosystem.

The Strategic Importance of ONNX

ONNX is an open standard for representing machine learning models. Its primary strategic value lies in its interoperability; it allows a model trained in one framework (like PyTorch) to be executed in a completely different environment (like a C# application) without dependencies on the original training stack.[4] This decoupling is essential for enterprise AI.
ONNX Runtime is a cross-platform, high-performance inference engine specifically designed to execute ONNX models with maximum efficiency.[6] It achieves this by abstracting the underlying hardware and leveraging platform-specific accelerators wherever possible, such as NVIDIA's CUDA for GPUs or DirectML on Windows.[4] By adopting ONNX, Aevenalytics can integrate the
multilingual-e5-large model directly into its existing C# services, benefiting from both the model's state-of-the-art NLP capabilities and the performance, stability, and security of the.NET platform.

Model Export and Validation

While many models on platforms like Hugging Face are available with pre-exported ONNX versions, relying on these artifacts without an internal validation process introduces a significant operational risk. The stability and compatibility of the .onnx file are mission-critical

for production deployment.
A documented failure case with a related model, intfloat/multilingual-e5-large-instruct, illustrates this risk perfectly. An automatically exported version of the model on Hugging Face used an ML opset (operator set) version that was incompatible with the latest version of ONNX Runtime.[8] The model's creators acknowledged they were not experts in ONNX and had relied on an automated export tool, leaving users with an unusable artifact.
This scenario demonstrates that Aevenalytics cannot assume third-party ONNX files will be correct, compatible, or maintained. To mitigate this risk, it is imperative to establish an in-house MLOps process for model validation and, when necessary, re-export. This process involves setting up a controlled Python environment with the necessary libraries (transformers, torch, onnx) and using tools like torch.onnx.export or Hugging Face's Optimum library to perform the conversion.[9] During this export, the
opset_version must be explicitly set to a version that is certified as compatible with the target version of the Microsoft.ML.OnnxRuntime C# library being used in production.[9] This internalizes control over a critical dependency, preventing deployment failures caused by upstream tooling issues or incompatible, unmaintained model artifacts.

C# Implementation Pipeline

The following steps outline a robust pipeline for using the exported multilingual-e5-large ONNX model in C# with the Microsoft.ML.OnnxRuntime library. This synthesized approach adapts best practices from various ONNX C# examples for models like BERT and Stable Diffusion.[7]

1. **Tokenization:** The first step is to convert input text into the numerical format the model expects. The multilingual-e5-large model uses a tokenizer based on XLM-RoBERTa.[1] This process must be replicated precisely in C#. While some specialized models can use pre-packaged ONNX-based tokenizers, a more general and reliable approach is to implement the tokenizer logic directly in C#. This involves using the model's vocabulary file (
   vocab.txt) and merges file (merges.txt) to map strings to sequences of token IDs. The open-source AllMiniLML6v2Sharp project provides a valuable C# reference for implementing a BERT-style tokenizer by loading a vocabulary file.[11] The output of this stage for each input string is two arrays:
   input_ids (the token IDs) and an attention_mask (a binary mask of 1s and 0s indicating which tokens are real and which are padding).
2. **Session and Input Tensor Creation:** An InferenceSession object is the primary interface for interacting with the ONNX model. It is initialized with the path to the .onnx file.[9] Creating this session is a computationally expensive operation, as it involves loading and parsing the model graph and preparing it for execution. Therefore, the InferenceSession should be instantiated once and cached for the lifetime of the application, for example, as a singleton service in a dependency injection framework.

For each inference call, the input_ids and attention_mask arrays are converted into OrtValue tensors. The OrtValue API is the recommended approach as it is designed for high performance, minimizing memory allocations and garbage collection overhead compared to older APIs.[13]

C#

```
// C# Pseudocode for Input Preparation
// Assumes 'tokenizer' is an initialized C# tokenizer instance
// Assumes 'session' is a cached InferenceSession instance

// 1. Tokenize the input text
(long inputIds, long attentionMask) = tokenizer.Encode(prefixedText);

// 2. Define tensor shapes
var inputIdsShape = new long { 1, inputIds.Length }; // Batch size of 1
var attentionMaskShape = new long { 1, attentionMask.Length };

// 3. Create OrtValue tensors from the tokenized data
using var inputIdsOrtValue = OrtValue.CreateTensorValueFromMemory(inputIds,
inputIdsShape);
using var attMaskOrtValue = OrtValue.CreateTensorValueFromMemory(attentionMask,
attentionMaskShape);

// 4. Prepare the dictionary of inputs for the session
var inputs = new Dictionary<string, OrtValue>
{
   { "input_ids", inputIdsOrtValue },
   { "attention_mask", attMaskOrtValue }
};
```

3. **Inference Execution:** The session.Run() method executes the model. It takes the dictionary of input OrtValues and returns a disposable collection of output OrtValues.[9] For the E5 model, the primary output of interest is the last_hidden_state, which is a tensor containing the embeddings for every token in the input sequence. Its shape will be [batch_size, sequence_length, embedding_dimension], which in this case is [1, sequence_length, 1024].
4. **Attention-Masked Mean Pooling:** This is a critical post-processing step that must be implemented manually in C#. The ONNX model itself only provides token-level embeddings; it does not produce a single, fixed-size sentence embedding.[10] To create this, a pooling operation is required. The standard for sentence transformers is mean pooling, but it must be performed correctly by only averaging the embeddings of the actual, non-padded tokens.
   The ONNX specification includes operators like AveragePool and GlobalAveragePool,

but these are designed for spatial pooling in computer vision models and are not suitable for this task.[14] They lack the capability to accept a secondary attention_mask tensor to selectively exclude certain vectors from the average calculation. Applying a standard pooling operator would incorrectly include the padded tokens (which have been zeroed out by the attention mechanism inside the model), thus corrupting the final sentence embedding by skewing it towards the origin.
Therefore, the correct masked mean pooling logic must be implemented in C# by directly manipulating the tensor data after inference. The logic, which mirrors the standard implementation in PyTorch [1], is as follows:

1. Extract the last_hidden_state tensor data as a Span<float> or array. This will be a flat array of size sequence_length * 1024.
2. Create an expanded attention mask. The original attention_mask has a shape of [1, sequence_length]. This needs to be broadcast or expanded to match the shape of the last_hidden_state tensor ([1, sequence_length, 1024]). In practice, this means for each token, its mask value (1 or 0) is applied to all 1024 dimensions of its embedding.
3. Perform an element-wise multiplication of the last_hidden_state embeddings with the expanded attention mask. This operation effectively sets the embedding vectors of all padded tokens to zero while leaving the real token embeddings unchanged.
4. Sum the resulting embeddings along the sequence dimension. This aggregates the values for each of the 1024 dimensions across all non-padded tokens, resulting in a single vector of size 1024.
5. Calculate the sum of the original attention_mask. This gives the exact count of non-padded tokens.
6. Perform an element-wise division of the summed embedding vector by the token count. This final step computes the true average and yields the correct sentence-level embedding.

5. **Normalization:** As a final step, the resulting 1024-dimensional vector should be L2-normalized. This process scales the vector so that its Euclidean length becomes 1. Normalization is a standard practice for sentence embedding models used in similarity tasks.[3] It ensures that the geometric distance between vectors in the embedding space is consistent and allows for the use of the computationally efficient dot product for calculating similarity, as it becomes mathematically equivalent to cosine similarity for unit vectors.

Performance Optimization

While powerful, the multilingual-e5-large model is computationally intensive. To ensure performance and cost-effectiveness in a production environment, Aevenalytics should investigate model quantization. Quantization is the process of reducing the precision of the

model's weights, for example, from 32-bit floating point (FP32) to 16-bit floating point (FP16) or 8-bit integer (INT8).
This process significantly reduces the model's file size and memory footprint. More importantly, it can lead to substantial speedups in inference time, particularly on CPUs, as integer arithmetic is much faster than floating-point arithmetic.[2] Tools like Hugging Face's Optimum library can be used to perform this quantization and export the optimized ONNX model.[10] The
Microsoft.ML.OnnxRuntime C# library fully supports running these quantized models. Aevenalytics should conduct a thorough benchmarking analysis to evaluate the trade-off between the performance gains from quantization and any potential minor degradation in embedding quality to determine the optimal precision for their specific application requirements.

Part II: Automated Knowledge Extraction and Graph Population

With a robust, high-performance pipeline for generating semantic embeddings, the next stage is to leverage this capability to automatically extract structured knowledge from unstructured text. This process involves identifying the most salient concepts—keyphrases—within discrete text chunks and using them to populate the knowledge graph with its initial set of nodes. This section provides a comparative analysis of leading keyphrase extraction methodologies and proposes a hybrid implementation strategy designed to maximize both the precision and the conceptual richness of the resulting graph.

2.1. A Comparative Analysis of Keyphrase Extraction Methodologies

The field of automated keyphrase extraction has progressed significantly from early statistical methods based on term frequencies (like TF-IDF or YAKE!) to more sophisticated approaches that leverage deep semantic understanding.[19] For a state-of-the-art system, the choice lies between two primary paradigms: embedding-based similarity and generative extraction using Large Language Models (LLMs).

Method 1: Embedding Similarity (KeyBERT)

The core principle of KeyBERT is both elegant and powerful. It operates by finding the phrases within a document that are most semantically representative of the document as a whole.[19] The process is as follows:

1. **Candidate Generation:** A set of candidate keyphrases is generated from the source

text, typically by extracting all n-grams within a specified length range (e.g., 1- to 5-word phrases).

2. **Embedding:** A sentence transformer model—in this case, the multilingual-e5-large model established in Part I—is used to generate a single embedding vector for the entire source document. The same model is then used to generate an embedding for each of the candidate n-grams.
3. **Similarity Scoring:** The cosine similarity is calculated between the document's embedding and the embedding of each candidate phrase.
4. **Selection:** The candidate phrases with the highest cosine similarity scores are selected as the final keyphrases. The underlying assumption is that the phrases whose meanings are closest to the meaning of the entire document are the most important.[23]

The primary strength of this method is its high degree of semantic relevance, which goes far beyond simple word counting. It leverages the nuanced understanding of the E5 model to identify conceptually important phrases. Furthermore, it is computationally efficient relative to LLM-based approaches and, crucially, guarantees that all extracted keyphrases are verbatim excerpts from the source text. This provides perfect traceability, which is essential for a verifiable knowledge graph. Its main limitation is that it can only extract what is explicitly stated; it cannot synthesize or abstract concepts that are implied but not written.

Method 2: LLM-based Generation (KeyLLM & Zero-Shot Prompting)

This paradigm shifts from extraction to generation, utilizing the advanced reasoning and language production capabilities of a Large Language Model.[24] By providing the source text as context within a carefully crafted prompt, an LLM can be instructed to generate a list of relevant keyphrases. This approach has two primary modes of operation:

1. **Extraction Mode:** The LLM is given a constrained prompt that instructs it to identify and return only the most important keywords and phrases that appear directly in the provided text.[24] This mode combines the semantic understanding of the LLM with the verbatim constraint of KeyBERT. The prompt might look like:
   "Based on the document below, extract the keyphrases that best describe the topic. Ensure you only extract phrases that appear in the text. Document:".
2. **Creation Mode:** The LLM is given a more open-ended prompt that asks it to generate keywords or themes that summarize the document, without the constraint that they must appear verbatim.[24] This mode allows the LLM to perform abstraction. For example, given a text describing shipping delays, component shortages, and factory shutdowns, the LLM could generate the abstract concept "supply chain disruption," even if those specific words never appear together.

The principal advantage of the LLM-based approach is this ability to abstract and generate higher-level, conceptual knowledge that is often missed by purely extractive methods. However, this comes at the cost of significantly higher computational overhead and latency. It also introduces the risk of model hallucination in creation mode and creates a dependency on

a separate, large-scale generative model and its associated API or hosting infrastructure.

A Hybrid Approach for Optimal Precision and Abstraction

Neither the embedding similarity method nor the LLM generation method is universally superior; they are optimized for different facets of knowledge capture. KeyBERT excels at the high-fidelity, traceable extraction of *explicit* concepts present in the text. LLMs excel at the reasoning-based generation of *implicit* or abstract themes that summarize the text.
A system that relies solely on KeyBERT would build a graph rich in precise, verifiable details but might miss the broader thematic connections between documents. Conversely, a system relying solely on an LLM would be slow, expensive, and might fail to capture important, specific terminology while introducing the risk of non-traceable, generated concepts.
Therefore, the optimal architecture for Aevenalytics is a hybrid, two-pass system that leverages the strengths of both approaches. This tiered strategy allows for a comprehensive capture of knowledge while effectively managing computational costs.

- **Pass 1 (Extraction):** The first pass will use the highly efficient KeyBERT methodology, powered by the already-implemented multilingual-e5-large embedding pipeline. This will serve as the workhorse of the system, rapidly processing all incoming document chunks to extract a foundational layer of salient, verbatim keyphrases. These form the explicit, high-precision nodes of the knowledge graph.
- **Pass 2 (Abstraction):** The second pass will use an LLM in "creation mode" in a more targeted fashion. Instead of running on every single document chunk, this more expensive process can be reserved for specific use cases, such as generating a high-level summary theme for an entire document or for a cluster of semantically related chunks that have been identified by the system. This adds a valuable layer of abstract, conceptual understanding to the graph without incurring prohibitive costs.

2.2. Proposed Implementation: The Keyphrase-to-Node Pipeline

The end-to-end process for extracting knowledge and populating the graph will follow this sequence:

1. **Input:** The process begins with a single, discrete text chunk provided by the MarkdownStructureChunker.
2. **Candidate Generation:** A list of all possible candidate phrases is generated from the chunk's text by extracting all n-grams up to a reasonable length (e.g., where n is between 1 and 5).
3. **KeyBERT Extraction (Pass 1):**
    - A single document-level embedding is generated for the entire text chunk using the C# E5 pipeline from Part I.
    - Embeddings are generated for every candidate phrase from the previous step.
    - The cosine similarity between the chunk's embedding and each candidate's

embedding is calculated.
    - The top-k candidates that exceed a predefined similarity threshold are selected as the extracted keyphrases.
4. **Node Creation:** For each keyphrase successfully extracted in the previous step:
    - A new node is created in the knowledge graph. The node should be assigned a primary label, such as ExtractedConcept.
    - The text of the keyphrase is stored as a primary property of the node (e.g., name).
    - Crucially, the 1024-dimensional E5 embedding of the keyphrase is also stored as a vector property on the node. This pre-computed embedding is essential for enabling efficient semantic search, clustering, and cross-lingual linking later.
    - Metadata properties are added to the node to maintain provenance, linking it back to the specific sourceChunkId from which it was extracted.
5. **LLM Abstraction (Pass 2 - Optional/Targeted):**
    - For specific high-value documents or clusters, the full text is sent to a generative LLM.
    - The prompt is engineered for abstraction, for example: "Analyze the following text and generate up to three abstract themes that summarize its core topic. Respond with the themes as a comma-separated list. Text:".
    - For each theme returned by the LLM, a new node is created in the graph with a distinct label, such as Theme. Its E5 embedding is generated and stored, and it is linked back to the source document(s).

This hybrid pipeline ensures that the knowledge graph is built on a foundation of high-precision, traceable data, which is then enriched with a layer of powerful, AI-generated conceptual abstraction.

Table 1: Keyphrase Extraction Method Comparison

To provide a clear, at-a-glance justification for the recommended hybrid approach, the following table compares the candidate methodologies across key operational and performance dimensions.

Method	Underlying Principle	Semantic Accuracy	Speed / Latency	Computational Cost	Traceability
YAKE! [21]	Statistical (word co-occurrence, position, casing)	Low-Medium	Very High	Very Low	Verbatim
KeyBERT [19]	Embedding Similarity (semantic	High	High	Low-Medium	Verbatim

	closeness to document)				
KeyLLM (Extract) [24]	LLM-Guided Extraction (reasoning over text)	Very High	Low	High	Verbatim
KeyLLM (Create) [24]	LLM Generation (conceptual abstraction)	High (can abstract)	Low	High	Abstract

This comparison clearly illustrates the trade-offs. While statistical methods like YAKE! are extremely fast and cheap, their semantic understanding is limited. At the other end of the spectrum, LLM-based creation offers powerful abstraction but at a significant cost in speed and resources. KeyBERT occupies a sweet spot, providing high semantic accuracy and perfect traceability with moderate computational cost. The proposed hybrid approach strategically combines the "KeyBERT" and "KeyLLM (Create)" methods to capture the benefits of both while mitigating their respective weaknesses.

Part III: Imposing Structure - The Ontology-Driven Graph Schema

A collection of extracted concepts, even if semantically rich, does not constitute a true knowledge graph. It is the formal, explicit definition of the types of entities and the relationships between them—the ontology—that transforms a data repository into a powerful system capable of consistency, inference, and complex reasoning.[27] An ontology serves as the schema for the graph, providing a shared vocabulary and a set of rules that govern its structure. This section outlines the core principles for designing a robust, domain-specific ontology for Aevenalytics and proposes a concrete starter schema to guide implementation.

3.1. Core Principles of Domain-Specific Ontology Engineering

The design of a useful and maintainable ontology is a structured engineering discipline. Drawing on best practices from complex, knowledge-intensive fields such as law provides a strong foundation for this work.[27] The following principles should guide the development of the Aevenalytics ontology.

Principle 1: User-Centric and Competency-Driven Design

The ultimate measure of an ontology's success is its utility. An effective ontology is not one that is theoretically perfect but one that enables the knowledge graph to answer the questions that its users need to ask.[27] Therefore, the design process must begin not with data, but with questions. This involves formally defining a set of **Competency Questions (CQs)**—concrete, natural language queries that the final system is expected to answer (e.g., "What are the most common issues reported for Product X?", "Which technical documents mention both Component Y and Failure Mode Z?"). These CQs serve as the functional requirements for the ontology; the necessary classes, properties, and relationships are precisely those that are needed to represent the information required to answer these questions.[30]

Principle 2: A Layered and Modular Approach

Attempting to build a single, monolithic ontology for an entire domain is a common anti-pattern that leads to brittle, unmanageable systems. A superior, more robust strategy is to adopt a layered and modular design.[29] This approach separates concerns and promotes reusability:

- **Foundational Ontology (Top-Level):** At the highest level of abstraction, the ontology should be grounded in a well-established foundational ontology. These provide a set of domain-agnostic, logically rigorous concepts like *Event*, *Agent*, *Object*, *Time*, and *Place*. Reusing a model like the Unified Foundational Ontology (UFO) or DOLCE+ ensures internal consistency and provides a stable base upon which to build.[30]
- **Core Ontology:** This middle layer defines concepts that are broadly applicable across the Aevenalytics enterprise but are more specific than the foundational layer. Examples might include Product, Customer, InternalProcess, or TechnicalSpecification.
- **Domain Ontology:** This is the most specific layer, containing the detailed classes, subclasses, and relationships that model the particulars of Aevenalytics' specific business units, product lines, and operational concepts.

Principle 3: Reuse of Ontology Design Patterns (ODPs)

Many modeling challenges are not unique. Rather than reinventing the wheel, the ontology should leverage established Ontology Design Patterns (ODPs) to represent common situations.[30] ODPs are reusable solutions to recurrent modeling problems. For example, the Agent-Role-Time pattern is a standard way to model that a person (Agent) acted in a specific capacity (Role) during a particular period (Time). Similarly, the Event-Time-Place pattern is used to situate occurrences. By adopting these patterns, development is accelerated, and the resulting ontology is more likely to be interoperable with other systems.

3.2. A Proposed Starter Ontology for the Aevenalytics Domain

To move from abstract principles to concrete implementation, a starter ontology is required. A highly effective model for this purpose can be adapted from the Lynx Legal Knowledge Graph (LKG) Ontology.[35]
The Lynx Ontology was designed to represent knowledge extracted from legal documents, but its core structure is brilliantly suited for any system that extracts concepts from structured text. It provides a clean, logical separation between the source text, its metadata, its structural components, and the semantic annotations (extracted concepts) derived from it. Aevenalytics' system, which begins with chunks from a MarkdownStructureChunker, maps perfectly to this paradigm. The document chunks are analogous to LynxDocumentPart, the extracted keyphrases are analogous to LynxAnnotation, and the associated file information is analogous to Metadata. The properties defined within Lynx, such as dct:language, lkg:part, and nif:referenceContext, are general-purpose descriptors for structured content and can be adapted directly. This provides a proven, robust foundation, saving significant design effort.

Proposed Classes:

- **AevenalyticsDocument**: Represents the original source document from which knowledge is extracted. It is recommended to make this a subclass of a standard vocabulary term like schema.org/CreativeWork for future interoperability.
    - **Properties:** sourceUrl (string), ingestionDate (datetime), documentType (string, e.g., 'TechnicalSpec', 'MeetingNotes').
- **DocumentChunk**: Represents an atomic, addressable block of text from the MarkdownStructureChunker. This is inspired by lkg:LynxDocumentPart.
    - **Properties:** chunkText (string), beginIndex (integer), endIndex (integer).
    - **Relationships:** hasParentDocument (points to AevenalyticsDocument), hasParentChunk (points to another DocumentChunk to model nested structures).
- **ExtractedConcept**: The primary node type for knowledge extracted from text, inspired by lkg:LynxAnnotation.
    - **Properties:** name (string, the keyphrase text), embeddingVector (1024-dim float array, the E5 embedding), extractionMethod (string, e.g., 'KeyBERT', 'KeyLLM').
- **NamedEntity**: A subclass of ExtractedConcept for specific, typed entities that can be identified through Named Entity Recognition (NER).
    - **Subclasses:** Person, Organization, Product, Location.

Proposed Properties (Relationships):

- **mentions**: The fundamental link between text and knowledge.
    - **Domain:** DocumentChunk

    - **Range:** ExtractedConcept
- **isA** (or rdfs:subClassOf): Used to create taxonomic hierarchies between concepts.
    - **Example:** A node `` would have an isA relationship to a [Product] node.
- **dependsOn**: A domain-specific relationship to model technical or procedural dependencies.
    - **Example:** [Firmware v3.2] dependsOn [HardwareModule v1.7].
- **sameAs** (or owl:sameAs): The critical relationship for unifying knowledge. It links nodes that represent the same real-world concept, particularly those identified across different languages. This will be populated by the methods in Part IV.

Table 2: Aevenalytics Starter Ontology Schema

The following table provides a formal specification for the proposed starter ontology. This serves as a concrete blueprint for data engineers and developers, ensuring a shared and unambiguous understanding of the knowledge graph's structure.[27]

Element	Type	Domain	Range	Description	Example
AevenalyticsDocument	Class	-	-	A source document ingested into the system.	A node representing internal_spec_v3.pdf.
DocumentChunk	Class	-	-	An atomic block of text from a source document.	A node representing lines 52-68 of internal_spec_v3.pdf.
ExtractedConcept	Class	-		A keyphrase or theme extracted from a DocumentChunk.	A node for the concept "luminous brew technology".
NamedEntity	Class	ExtractedConcept	-	A subclass for specific, typed entities.	A node for the product "GlowBrew".
sourceUrl	Data Property	AevenalyticsDocument	string	The URI or path of the original source file.	file:///specs/internal_spec_v3.pdf
embeddingVector	Data Property	ExtractedConcept	float	The 1024-dimensional E5 embedding of	[0.12, -0.45,...]

				the concept's name.	
hasParentDocument	Object Property	DocumentChunk	AevenalyticsDocument	Links a chunk to its source document.	(Chunk 52-68) -> hasParentDocument -> (spec_v3.pdf)
mentions	Object Property	DocumentChunk	ExtractedConcept	Links a chunk to a concept it contains.	(Chunk 52-68) -> mentions -> ("luminous brew technology")
isA	Object Property	ExtractedConcept	ExtractedConcept	Creates a taxonomic hierarchy.	(GlowBrew) -> isA -> (Product)
sameAs	Object Property	ExtractedConcept	ExtractedConcept	Links two nodes representing the same concept.	("LED array") -> sameAs -> ("matrice de LED")

3.3. Ontological Enforcement in Graph Construction

The ontology is not merely a design document; it is an active schema that must be enforced during the graph construction process. As the pipelines from Part II extract keyphrases and create nodes and relationships, the graph database or middleware layer must validate each new addition against the rules defined in the ontology.[36] For example, the system should reject any attempt to create a
mentions relationship originating from an AevenalyticsDocument node, as the ontology specifies that its domain is strictly DocumentChunk. This enforcement ensures the graph's logical consistency, prevents data corruption, and guarantees that the knowledge contained within is clean, reliable, and ready for high-level reasoning and querying.

Part IV: Unifying Knowledge Across Linguistic and Contextual Divides

The final and most advanced architectural component is the system's ability to create a unified knowledge graph that transcends linguistic barriers. The goal is to ensure that a

concept extracted from a document in one language is formally linked to the same concept extracted from a document in another language. This creates a single, coherent semantic network where the language of origin is merely an attribute, not a silo. This section details a highly effective primary method for achieving this linkage and a state-of-the-art technique for advanced, graph-wide refinement.

4.1. Primary Method: Cross-Lingual Linking via Shared Embedding Space

The core mechanism for achieving cross-lingual unification is a direct and powerful application of the chosen embedding model, intfloat/multilingual-e5-large. By its very design, this model projects semantically equivalent concepts into the same region of its 1024-dimensional vector space, regardless of the source language in which those concepts were expressed.[1] A concept like "protein requirement" in English and its Chinese equivalent will have embedding vectors that are extremely close to each other. This property is the key enabler for automated cross-lingual linking.

Implementation Workflow

The process for discovering and creating these cross-lingual links is integrated directly into the node creation pipeline:

1. **Embedding Storage:** As established in Part II, every ExtractedConcept node created in the knowledge graph must have its 1024-dimensional E5 embedding stored as a vector property on the node itself. This is a prerequisite for any semantic operations.
2. **Nearest Neighbor Search:** When a new concept is extracted from a document (e.g., the Chinese keyphrase "南瓜的家常做法" from [1], meaning "homestyle pumpkin recipes"), the system performs a query against the knowledge graph. This query is not a text search but an
   Approximate Nearest Neighbor (ANN) search. The new concept's embedding vector is used to find the existing nodes in the graph whose stored embedding vectors are closest in the vector space. Modern graph databases and vector search libraries are highly optimized for this type of query, making it scalable to millions of nodes.
3. **Similarity Thresholding:** The ANN search will return a ranked list of the most similar existing concepts. The system then evaluates the cosine similarity between the new concept's embedding and the top search results. If a match is found that exceeds a very high similarity threshold (e.g., a cosine similarity greater than 0.95), the two concepts are considered to be semantically identical.
4. **Link Creation:** Upon finding a confident match, a new relationship is created in the graph. An edge with the label sameAs (from the ontology in Part III) is created, linking the new node to the existing synonymous node.

This workflow effectively uses the shared embedding space to discover latent relationships between terms that may have never appeared together in any document. The linkage is based purely on their shared meaning, as understood by the powerful multilingual model. This directly and efficiently fulfills the core requirement to "connect domain-specific terms across languages," creating a web of equivalences that unifies the graph.

4.2. Advanced Refinement: Graph Alignment with Joint Embedding Models

While the direct embedding similarity approach is highly effective and scalable for real-time linking, its perspective is localized; it compares concepts on a one-to-one basis. A more advanced and holistic approach can be employed to refine these linkages by considering the entire structure of the graph. Academic research in cross-lingual entity alignment has produced sophisticated models for this purpose, which can be adapted for periodic, graph-wide refinement.[37]

Introducing JAPE (Joint Attribute-Preserving Embedding)

The Joint Attribute-Preserving Embedding (JAPE) model represents a state-of-the-art methodology for aligning two distinct knowledge graphs.[37] It operates by learning a unified embedding space where entities from both graphs are positioned based not only on their intrinsic properties but also on the structure of their relationships with other entities.

Strategic Application for Aevenalytics

For Aevenalytics, instead of aligning two entirely separate knowledge bases, the JAPE methodology can be cleverly adapted to align different language-specific *subgraphs* within the main knowledge graph. For instance, the collection of all nodes and relationships derived from English-language documents can be treated as one graph, and the collection derived from Chinese-language documents as another. The JAPE model can then be trained to find and strengthen the sameAs links between them.
The model works through two complementary modules:

- **Structure Embedding (SE):** This module learns from the patterns of relationships (triples) in the graph. For example, if the English subgraph contains the triple (ConceptA, dependsOn, ConceptB) and the Chinese subgraph contains (概念A, 依赖于, 概念B), the SE module learns that dependsOn and 依赖于 are similar relationships. This, in turn, reinforces the likelihood that ConceptB and 概念B are also the same entity. The initial sameAs links created by the E5 similarity search serve as the crucial "seed alignment" that the JAPE model needs to begin this learning process.

- **Attribute Embedding (AE):** This module leverages the properties (attributes) of the nodes. If two concept nodes, one from the English subgraph and one from the Chinese, share similar attributes (e.g., they were both extracted from documents with the property documentType: 'TechnicalSpec', or they both have a high frequency of co-occurrence with another aligned entity pair), the AE module increases their similarity score.

Due to its computational intensity, the JAPE model is not suited for real-time, transactional linking. Instead, it should be implemented as a periodic batch process that runs offline (e.g., weekly). This process would ingest the current state of the graph, train the alignment model, and then use the model's predictions to validate, correct, and discover new sameAs links. It serves as a powerful, graph-aware refinement and auditing mechanism, using the holistic context of the entire knowledge graph to improve the accuracy of the fine-grained, cross-lingual connections established by the primary E5-based method.

Conclusion and Strategic Roadmap

The architecture detailed in this report provides a comprehensive blueprint for Aevenalytics to construct a state-of-the-art, ontology-driven, multilingual knowledge graph. This system is designed to transform unstructured text into a structured, queryable, and semantically rich knowledge asset. By integrating a high-performance multilingual embedding pipeline, a hybrid keyphrase extraction strategy, a formal ontological schema, and advanced cross-lingual linking techniques, the proposed architecture will create a durable and scalable foundation for a new generation of intelligent applications.

The successful implementation of this vision requires a phased approach, allowing for iterative development, testing, and value delivery. The following strategic roadmap is recommended:

Phase 1: Foundational Implementation (Months 1-3)

The initial phase focuses on building the core data processing and ingestion pipeline. The primary goal is to establish the capability to convert text into embedded concepts and populate an initial version of the graph.

- **Task 1: Model Validation and Deployment Pipeline:** Establish the Python-based MLOps process for validating and exporting the intfloat/multilingual-e5-large model to a compatible ONNX format. This is a critical risk mitigation step.
- **Task 2: C# Embedding Service:** Develop the core C# service using ONNX Runtime. This includes implementing the tokenizer, session management, and the crucial custom logic for attention-masked mean pooling and vector normalization.
- **Task 3: Keyphrase Extraction and Node Creation:** Implement the KeyBERT-based extraction pipeline (Pass 1). This will be the primary mechanism for populating the graph with explicit, traceable concept nodes, each with its stored E5 embedding.

Phase 2: Ontology and Structure (Months 4-6)

With the foundational ingestion pipeline in place, the focus shifts to imposing a formal structure on the graph and enabling cross-lingual connections.

- **Task 1: Competency Question Workshops:** Conduct internal workshops with key stakeholders to define and prioritize the initial set of "competency questions" that the knowledge graph must answer. This will guide the final ontology design.
- **Task 2: Ontology Formalization:** Formalize the starter ontology proposed in this document (Table 2) into a machine-readable format (e.g., RDFS/OWL) and integrate it into the graph database system as a schema.
- **Task 3: Ontology-Driven Ingestion:** Refactor the graph population pipeline to be ontology-aware, ensuring all new nodes and edges are validated against the schema before being committed.
- **Task 4: Initial Cross-Lingual Linking:** Implement the primary cross-lingual linking method using Approximate Nearest Neighbor (ANN) search on the stored E5 embeddings to create the initial set of sameAs relationships between concepts from different languages.

Phase 3: Advanced Capabilities and Refinement (Months 7-12)

The final phase builds upon the structured graph to introduce more advanced knowledge extraction and refinement capabilities, preparing the system for user-facing applications.

- **Task 1: LLM-based Abstraction:** Integrate the LLM-based abstraction pipeline (Pass 2). This involves setting up the necessary infrastructure to call a generative LLM and designing prompts to generate high-level thematic concepts for high-value documents or document clusters.
- **Task 2: Graph Alignment Refinement:** Develop and deploy the JAPE-based graph alignment model as a periodic, offline batch process. This will serve to audit and refine the accuracy of the sameAs cross-lingual links over time.
- **Task 3: Application Layer Development:** Begin the development of the first consumer applications that will leverage the knowledge graph, such as an advanced semantic search engine, a question-answering system, or a knowledge exploration interface.

By following this phased roadmap, Aevenalytics can systematically build a powerful and strategic knowledge asset. The resulting semantic core will not only address the immediate requirements but will also provide a flexible and extensible platform to drive innovation and unlock deeper insights from the company's vast textual data resources for years to come.

Citerede værker

1. intfloat/multilingual-e5-large - Hugging Face, tilgået september 16, 2025,

https://huggingface.co/intfloat/multilingual-e5-large
2. Vespa Onnx Intfloat Multilingual E5 Large · Models - Dataloop, tilgået september 16, 2025, https://dataloop.ai/library/model/hotchpotch_vespa-onnx-intfloat-multilingual-e5-large/
3. SentenceTransformer — Sentence Transformers documentation, tilgået september 16, 2025, https://sbert.net/docs/package_reference/sentence_transformer/SentenceTransformer.html
4. onnxruntime - ONNX Runtime, tilgået september 16, 2025, https://onnxruntime.ai/docs/
5. microsoft/onnxruntime: ONNX Runtime: cross-platform, high performance ML inferencing and training accelerator - GitHub, tilgået september 16, 2025, https://github.com/microsoft/onnxruntime
6. Generate images with AI using Stable Diffusion, C#, and ONNX Runtime - .NET Blog, tilgået september 16, 2025, https://devblogs.microsoft.com/dotnet/generate-ai-images-stable-diffusion-csharp-onnx-runtime/
7. Inference Stable Diffusion with C# and ONNX Runtime, tilgået september 16, 2025, https://onnxruntime.ai/docs/tutorials/csharp/stable-diffusion-csharp.html
8. ONNX model of intfloat / multilingual-e5-large-instruct using unsupported "Onnx ML opset" 5 - can't load - Hugging Face, tilgået september 16, 2025, https://huggingface.co/intfloat/multilingual-e5-large-instruct/discussions/22
9. Inference BERT NLP with C# | onnxruntime, tilgået september 16, 2025, https://onnxruntime.ai/docs/tutorials/csharp/bert-nlp-csharp-console-app.html
10. Speeding up Inference — Sentence Transformers documentation, tilgået september 16, 2025, https://sbert.net/docs/sentence_transformer/usage/efficiency.html
11. ksanman/AllMiniLML6v2Sharp: net standard 2.1 Library for ... - GitHub, tilgået september 16, 2025, https://github.com/ksanman/AllMiniLML6v2Sharp
12. C# | onnxruntime, tilgået september 16, 2025, https://onnxruntime.ai/docs/get-started/with-csharp.html
13. Basic C# Tutorial | onnxruntime, tilgået september 16, 2025, https://onnxruntime.ai/docs/tutorials/csharp/basic_csharp.html
14. AveragePool - ONNX 1.20.0 documentation, tilgået september 16, 2025, https://onnx.ai/onnx/operators/onnx__AveragePool.html
15. GlobalAveragePool — ONNX 1.12.0 documentation, tilgået september 16, 2025, https://natke.github.io/onnx/operators/onnx__GlobalAveragePool.html
16. optimum/all-MiniLM-L6-v2 - Hugging Face, tilgået september 16, 2025, https://huggingface.co/optimum/all-MiniLM-L6-v2
17. Understanding Embeddings with ModernBERT: Comparing Mean Pooling via Transformers and SentenceTransformers | by Ismailvanak | Medium, tilgået september 16, 2025, https://medium.com/@ismailvanak/understanding-embeddings-with-modernbert-comparing-mean-pooling-via-transformers-and-300ef0d6b87a

18. How to compute mean/max of HuggingFace Transformers BERT token embeddings with attention mask? - Stack Overflow, tilgået september 16, 2025, https://stackoverflow.com/questions/65083581/how-to-compute-mean-max-of-huggingface-transformers-bert-token-embeddings-with-a
19. MaartenGr/KeyBERT: Minimal keyword extraction with BERT - GitHub, tilgået september 16, 2025, https://github.com/MaartenGr/KeyBERT
20. Keyword Extraction with BERT - Maarten Grootendorst, tilgået september 16, 2025, https://www.maartengrootendorst.com/blog/keybert/
21. Complete Guide to Keywords/Phrase Extraction - Kaggle, tilgået september 16, 2025, https://www.kaggle.com/code/akashmathur2212/complete-guide-to-keywords-phrase-extraction
22. Keyword Extraction Methods from Documents in NLP - Analytics Vidhya, tilgået september 16, 2025, https://www.analyticsvidhya.com/blog/2022/03/keyword-extraction-methods-from-documents-in-nlp/
23. Two minutes NLP — Keyword and keyphrase extraction with KeyBERT | by Fabio Chiusano | Generative AI | Medium, tilgået september 16, 2025, https://medium.com/nlplanet/two-minutes-nlp-keyword-and-keyphrase-extraction-with-keybert-a9994b06a83
24. KeyLLM - KeyBERT - Maarten Grootendorst, tilgået september 16, 2025, https://maartengr.github.io/KeyBERT/guides/keyllm.html
25. Introducing KeyLLM - Keyword Extraction with LLMs - Maarten Grootendorst, tilgået september 16, 2025, https://www.maartengrootendorst.com/blog/keyllm/
26. Empirical Study of Zero-shot Keyphrase Extraction with Large Language Models - ACL Anthology, tilgået september 16, 2025, https://aclanthology.org/2025.coling-main.248.pdf
27. Ontologies for lawyers. At recent lawyer-focused hackathons and... | by Margaret Hagan | Legal Design and Innovation | Medium, tilgået september 16, 2025, https://medium.com/legal-design-and-innovation/ontologies-for-lawyers-5c3b9fb23439
28. Ontology in Graph Models and Knowledge Graphs, tilgået september 16, 2025, https://graph.build/resources/ontology
29. Legal Domain Ontologies, tilgået september 16, 2025, https://ontology.buffalo.edu/FARBER/visser.html
30. Capturing the Basics of the GDPR in a Well-Founded Legal Domain Modular Ontology - FOIS 2021, tilgået september 16, 2025, https://fois2021.inf.unibz.it/wp-content/uploads/2021/08/paper_8.pdf
31. (PDF) A Constructive Framework for Legal Ontologies - ResearchGate, tilgået september 16, 2025, https://www.researchgate.net/publication/225148250_A_Constructive_Framework_for_Legal_Ontologies
32. Ontologies for lawyers | A Better Legal Internet, tilgået september 16, 2025, https://betterinternet.law.stanford.edu/2019/10/18/ontologies-for-lawyers/
33. Natural Language Processing for the Legal Domain: A Survey of Tasks, Datasets,

Models, and Challenges - arXiv, tilgået september 16, 2025, https://arxiv.org/html/2410.21306v1
34. Legal Knowledge Extraction for Knowledge Graph Based Question-Answering, tilgået september 16, 2025, https://cris.unibo.it/retrieve/e1dcb336-1cbb-7715-e053-1705fe0a6cc9/FAIA-334-FAIA200858.pdf
35. Legal Knowledge Graph Ontology - Lynx, tilgået september 16, 2025, https://lynx-project.eu/doc/lkg/
36. Ontologies: Blueprints for Knowledge Graph Structures - FalkorDB, tilgået september 16, 2025, https://www.falkordb.com/blog/understanding-ontologies-knowledge-graph-schemas/
37. Cross-lingual Entity Alignment via Joint Attribute-Preserving ..., tilgået september 16, 2025, https://arxiv.org/abs/1708.05045
38. Neural Cross-Lingual Entity Linking - AAAI, tilgået september 16, 2025, https://cdn.aaai.org/ojs/11964/11964-13-15492-1-2-20201228.pdf
39. [1909.13180] Towards Zero-resource Cross-lingual Entity Linking - arXiv, tilgået september 16, 2025, https://arxiv.org/abs/1909.13180
40. [1712.01813] Neural Cross-Lingual Entity Linking - arXiv, tilgået september 16, 2025, https://arxiv.org/abs/1712.01813