ChatGPT

Evolving Semantic Graphs and Ontologies with LLMs and MGraph-DB

Introduction:
This FAQ-style white paper addresses key technical questions about Dinis Cruz's approach to building and evolving semantic knowledge graphs using Large Language Models (LLMs) and the **MGraph-DB** platform. The focus is on how this architecture differs from traditional Semantic Web frameworks (like OWL ontologies) and how it leverages MGraph-DB's unique design. We will discuss concept identification, confidence scoring, reasoning methods, graph lifecycle management, and the idea of an "ontology of ontologies." Each section below poses a question and provides an expanded answer, highlighting the professional and business value of this approach for a technically literate audience.

Q1: How does your semantic graph approach handle concept identifiers compared to OWL?

Handling of Concept Identifiers:
In a traditional OWL ontology, each concept is identified by a global IRI/URI – a strict, often lengthy identifier ensuring unambiguous reference across datasets. Dinis Cruz's semantic graph approach takes a more flexible and organic path in assigning concept identifiers. Instead of predefining every concept with a fixed global ID, the system allows LLMs to **propose concepts in natural language** and uses context or mappings to resolve their identities over time. Key differences include:

- **Human-Friendly Labels vs IRIs:** Concepts extracted by the LLM are usually labeled in plain language (e.g. "Cardiology" or "CEO persona") without immediately binding them to a formal URI. This makes initial graph construction quick and relatable. By contrast, OWL demands something like `<http://example.com/ontology#Cardiology>` upfront. In our graphs, each node does have a unique internal ID (for example, a UUID or auto-increment index), but it's used behind the scenes for linking; the emphasis is on readable labels that can later be aligned or refined.

- **Contextual Uniqueness:** The approach leverages context to distinguish concepts. Two teams might independently have a concept called "Asset" in their respective sub-graphs, and that's acceptable initially. We don't force a single canonical "Asset" definition at creation time. In OWL, by contrast, using the same name would imply the exact same concept (or you'd create separate IRIs). With MGraph-DB, each "Asset" node lives in its graph namespace, and if they truly represent the same real-world concept, a **mapping process** (often LLM-assisted) will later connect or merge them. Essentially, identity is established through relationships and mappings, not just by a globally unique name.

- **Emergent Alignment vs Pre-Coordinated Ontology:** Rather than predetermining a universal ontology of all concepts, Cruz's method lets the knowledge graph *emerge and evolve*. The LLM might introduce new concepts as needed when processing data (for example, a new relationship "specializes in [Field]" appears while analyzing an article). These are added to the graph on the fly. Later, during a curation phase, these concepts can be reconciled with existing ontology entries or given persistent identifiers if warranted. OWL systems tend to require the ontology's classes and properties to be designed upfront or continuously maintained by experts; here the ontology grows dynamically, guided by LLM output and human review.

- **No Strict IRI Schema Requirements:** MGraph-DB does not require using HTTP IRIs for nodes; it stores graph data in a JSON-based structure. If needed, it can store a concept's canonical reference (e.g. an external ontology ID) as an attribute, but it doesn't force every node to carry an IRI. This flexibility means quicker integration of diverse data sources: you can ingest information without painstakingly converting all identifiers to a single naming scheme. (The graph can always be exported to standard formats like RDF/Turtle with IRIs once things are aligned, but during iterative development the overhead is minimized.)

Implications:
This approach trades a bit of up-front formality for agility. It acknowledges that in practical, evolving systems, you often encounter the same idea described in different ways. Instead of insisting on one name from day one, it captures all the variations and then unifies them through analysis. **MGraph-DB** supports this by allowing duplicate or similar labels internally (since each node has its own GUID) and by making it easy to add equivalence links or merge nodes later. The result is a semantic graph that **starts with the language people actually use** and **systematically moves toward consolidation**, rather than starting with a rigid ontology that might not fit all future data. In summary, concept identifiers in this system are handled more fluidly than in OWL: they are **initially simple and contextual**, and they become **persistent and globally unique through iterative refinement and mapping**, rather than being fully predetermined.

Q2: How are the various confidence or relevance scores calculated and what do they mean?

Origin of Scores in the System:
Throughout the LLM-driven graph pipeline, the system attaches numeric scores to certain outputs – these can represent *confidence, relevance, or strength* of a relationship. Unlike deterministic ontologies where a fact is either present or not, an LLM-generated knowledge graph benefits from having a gradation of certainty or importance. Here's how these scores are produced and used:

- **LLM-Generated Relevance Ratings:** In many cases, the LLM is prompted not only to extract entities and relationships, but also to **assess the relevance or confidence** of those findings. For example, when linking a persona's interest graph to a set of articles, the LLM might be asked: "For each article, identify which persona topics it covers and give a relevance score from 0 to 10." A concrete example: the model might report that a particular article has *3 topic matches with a persona* and assign an **overall relevance score of 8.5/10**. This number is a qualitative judgment by the AI – essentially how strongly it believes the article aligns with the persona's interests. It could be influenced by the number of matching concepts and the significance the model perceives those concepts to have in context. The scale (often 0-1 or 0-10) is chosen in the prompt; 8.5/10 here would indicate a high relevance, whereas something like 3/10 would be only mildly relevant.

- **Confidence Scores on Extracted Facts:** Similarly, when the LLM extracts a fact or relationship, it can be instructed to include a confidence indicator. For instance, if it identifies an entity "Cloud Computing" as a key topic of an input text, it might output that with a **confidence of 0.95** (on a 0–1 scale). This would mean the model is 95% sure "Cloud Computing" is correctly identified. These confidence scores are generally subjective probabilities from the model's perspective.

They are not guarantees of truth, but they provide a useful heuristic. A lower confidence (e.g. 0.6) might flag an item for human review or for cross-checking against the knowledge base.

- **Algorithmic or Composite Scores:** Not all scoring is left purely to AI guesswork. In some parts of the workflow, scores are computed by deterministic logic based on graph data. For example, if multiple evidence sources in the knowledge base support a relationship, the system might aggregate that into a higher confidence. In threat modeling use-cases, one could imagine a “risk score” computed from connected nodes (number of incidents linked to a threat, plus impact level) – that’s a form of score derived from graph analytics. The key point is that the architecture supports storing these numeric attributes on nodes/edges, whether they come from an LLM’s qualitative judgment or from a script’s calculation.

Meaning and Usage of Scores:
A score is only as meaningful as the process behind it. In this system, scores are used to **prioritize and filter knowledge**:

- A high relevance score (say 8.5/10) for an article under a persona’s topic means that article is very likely to be selected for that persona’s curated feed or report. In the MyFeeds example, only the top-scoring articles (e.g. five articles with highest scores above a threshold) were chosen as the personalized digest, ensuring the user sees content most aligned with their interests. Here the score effectively **ranks content by pertinence**.

- Confidence scores serve as a guide for trust and verification. If an extracted relationship has low confidence, the system can mark it for a human-in-the-loop to verify or ignore it until further evidence is gathered. High-confidence facts might be ingested automatically into the knowledge base, whereas lower-confidence ones might need an additional LLM query or a rule-based check. In other words, the **graph evolves conservatively**, integrating only those facts that pass a confidence bar (much like a precision threshold) to maintain overall quality.

- Over time, these scores can be updated. If later data or user feedback confirms a once-low-confidence fact, its score can be raised. If a highly-scored relationship is later contradicted by new information (or turns out to be a hallucination), it can be rescored downwards or removed. Because the graphs are version-controlled (more on that later), any change in scores or filtered inclusion of nodes can be tracked and justified.

In summary, confidence and relevance scores in this system are **quantitative signals of certainty and importance** attached to graph elements. They are calculated either by the LLM (based on its internal assessment) or by auxiliary logic, and they mean to guide the **flow of information** – what gets emphasized, what gets secondary treatment, and what might require human confirmation. This adds an important layer of transparency and tunability to LLM-generated knowledge graphs: stakeholders can see *why* a piece of information was included (it had a high score), and they can decide to adjust the process (for instance, “only include articles with score >7” or “double-check any relationship below 0.8 confidence”). It’s a pragmatic way to blend the **fuzzy intuition of AI with the precision needs of a business**.

Q3: What kind of reasoning is used in your system? Do you rely on Description Logic like OWL, or take a stochastic/LLM-driven approach?

The reasoning in this architecture is largely **LLM-driven and pragmatic** rather than based on formal description logic. Traditional OWL ontologies utilize description logic reasoners to infer new facts (e.g., deducing class hierarchies, checking consistency, applying transitive properties automatically). In contrast, Dinis Cruz's system leans on the power of LLMs combined with lightweight code-based rules for reasoning. Here is how it works:

- **LLM as the Primary Reasoning Engine:** The heavy lifting of understanding context and proposing connections is done by the LLM (a large language model). This is a stochastic approach – meaning it uses probability and pattern-matching learned from data, not a guaranteed logical calculus. For example, if given a software architecture document, the LLM might *infer* that "Module A depends on Module B" because it reads between the lines, even if that dependency wasn't stated in a formal way. This kind of reasoning is implicit and generative. It's very powerful for uncovering links or insights that aren't explicit, but it doesn't follow strict logical proof. We accept that and mitigate risks (like possible errors) by capturing the LLM's output in structured form and then verifying it as needed.

- **No Built-in OWL Description Logic Reasoner:** We do not run a classical OWL reasoner on the graph to derive entailments. For instance, if the ontology says "Every X is a Y" and we have an X node, an OWL reasoner would automatically classify that node also as a Y. In our system, such hierarchical inferences would either need to be handled by the LLM (which could be instructed to output types for each entity) or by a simple code routine. We aren't using a DL reasoner to enforce or compute class relationships; instead, we maintain logical consistency through the design of our data model and validation tests. This approach is closer to how **knowledge graphs in industry** often work – they might adhere to an ontology schema, but they don't always run a reasoner continuously. It sacrifices some of the guaranteed completeness of a formal ontology in exchange for flexibility and performance.

- **Rule-Based and Deterministic Logic (Where Applicable):** Although we lean on LLMs, we complement them with **deterministic reasoning steps** for things that are well-defined. A good principle Dinis follows is: *"Only use LLMs for what they are uniquely good at, and use code for everything else."* That means if we need to enforce a rule like "every `User` node must have an `email` property," we don't ask the LLM to check that – we write a simple test in code. In practice, after the LLM generates the graph data, we might run a suite of **graph data tests** (similar to unit tests) to ensure logical constraints and business rules are satisfied. For example, a rule-based check could flag any node labeled "Company" that isn't linked to a "Sector" node (if our domain requires every company to be categorized). These kind of checks act as a lightweight reasoner, ensuring the graph doesn't violate known constraints. They're not as sophisticated as OWL DL inference, but they are **human-understandable and debuggable**.

- **Human-in-the-Loop Reasoning:** Another form of reasoning in the system comes from human experts reviewing and curating the output. If an LLM posits a relationship that "Technology X is a subtype of Strategy Y" and scores it somewhat low, a human can decide if that reasoning is sound. Essentially, some reasoning tasks (especially in complex domains like cybersecurity or finance) are deferred to subject matter experts who use the graphs and provide feedback. Their feedback may translate into new rules ("actually, X should never be subtype of Y, it should be related via `uses` relationship") or into corrections in the knowledge base. Over time, this **expert feedback** becomes part of the system's reasoning process, because the pipeline can incorporate those new rules or examples in future LLM prompts (making the AI reasoning more accurate in the next iteration).

- **Why Not Full Semantic Reasoners?:** It's worth noting why this approach avoids a full DL reasoner. Performance and determinism are part of the reason. MGraph-DB is designed as a **memory-first, JSON-backed graph store** with an emphasis on speed and integration with code. Running an OWL reasoner on a large graph can be slow and opaque, and doing so continuously in a dynamic, evolving graph scenario might be impractical. Instead, we achieve many of the benefits (like catching inconsistencies or inferring obvious connections) through **structured outputs and validations**. The LLM's structured output schema can be seen as a kind of reasoning template – for example, if we expect a `Person` node to have a `birthDate`, we include that in the schema so the LLM will try to fill it. This way the LLM is guided by a pseudo-ontology to output logically complete data. Then our code tests check nothing critical is missing or contradictory. This two-layer approach (AI + code) yields a system that is *deterministic in its processing pipeline* (you can rerun it on the same input and get the same structured result) even though the AI inside is stochastic.

In summary, the system favors an **LLM-driven approach to reasoning**, augmented by **deterministic, rule-based checks** and human oversight. It does not use OWL's description logic engines for automated inference; instead, it uses the intelligence of the LLM to propose links and the rigor of code to enforce clear rules. This results in a solution that is both **creative and flexible** (thanks to AI) and **controlled and explainable** (thanks to a transparent pipeline). For a business context, this means we get the best of both worlds: rapid, AI-generated insights and a clear audit trail of how conclusions were reached, without the black-box complexity that formal semantic reasoners can introduce.

Q4: How are the graphs produced by the LLM stored, managed, and evolved over time? What is the role of the knowledge base?

Storage in MGraph-DB (Knowledge Base):
Graphs produced by the LLM are not just ephemeral structures held in memory; they are persisted and managed in what we call the knowledge base. In this architecture, the knowledge base is essentially a **collection of versioned graph data** stored using MGraph-DB's format (JSON). Concretely, when an LLM finishes a task (say extracting entities and relationships from an article or generating a mapping between two ontologies), the result is saved as a JSON file (or set of files) representing a graph – nodes, edges, properties, and even the LLM-generated metadata like scores. Because MGraph-DB is file-backed and schema-aware, each JSON snapshot can be checked into Git or cloud storage. This means:

- Every iteration or update to the graph can be diffed and tracked (just like source code). The knowledge base therefore becomes a **living history of the graph's evolution**. You might have `PersonaGraph_v1.json`, `PersonaGraph_v2.json`, etc., showing how new nodes or edges were added by subsequent LLM runs or human edits.

- The knowledge base is **highly portable**. Since it's just structured data (e.g., JSON), we aren't locked into a single database engine or instance. A graph file can be loaded into an MGraph-DB instance on a developer's laptop, a serverless cloud function, or any environment with Python. This supports a serverless, on-demand usage pattern: you only load and instantiate the graph when you need to query or modify it. At other times, it rests as static data (costing nothing when not in use).

Management and Evolution:
Managing these graphs over time involves both automated processes and human-in-the-loop processes:

- **Deterministic Pipelines:** The architecture is set up as a pipeline of steps (similar to an ETL or CI/CD pipeline for data). For example:
- **Extraction Phase:** LLM reads raw input (RSS feed, document, etc.) and outputs an initial graph (saved as JSON).
- **Enrichment/Mapping Phase:** Another LLM call or a script takes those graphs and links them, adding mapping nodes or edges (output another JSON).
- **Curation Phase:** Optionally, a human reviews the combined graph and edits it (either via a UI or by editing the JSON directly or through a Git commit).
- **Utilization Phase:** The final graph is loaded to answer questions or to generate a report.

Each phase's output is stored, and because the pipeline is deterministic given the same inputs, we can always reproduce a graph version by rerunning the pipeline. This structure makes the graph evolution **repeatable and debuggable**. If a certain relationship appears incorrect, we can trace it to the stage (and even the prompt) that produced it and adjust that stage.

- **MGraph-DB as the Engine:** MGraph-DB plays a foundational role in this management. It serves as the in-memory graph engine when we need to manipulate or query the graph. It's optimized for fast in-memory operations, which means even as the knowledge base grows, we can spin up an MGraph-DB instance on a subset of the data (or the entire dataset) to run complex queries, analytics, or transformations efficiently. Once changes are made (e.g., merging two nodes or adding a new edge), we can serialize the graph back to JSON and commit the changes. There is no separate heavy database process running 24/7 – we use MGraph-DB in a **stateless manner**. It's analogous to how one might use a compiler: run it when needed, then store the artifact. Here, the artifact is the updated graph dataset.

- **Versioning and Snapshots:** Because everything is file-based, implementing version control (with Git or another VCS) is straightforward. Every change to the knowledge graph can go through a **GitOps style workflow**: propose changes (even via pull requests), review them (diff the JSON, perhaps with a visual diff tool for graphs), and merge them. This is a major departure from traditional graph databases where changes accumulate in a running instance and it's non-trivial to capture the exact state or roll back individual changes. Our approach treats knowledge as code, enabling **branching and experimentation**. For example, one could branch the knowledge base to try a different ontology alignment, and if it doesn't work out, simply not merge those changes.

- **Maintenance and Evolution Over Time:** Over time, as new data arrives or the world changes, the graphs need to be updated. The LLM might be run periodically (say, nightly or whenever new inputs are available) to extend the graphs. The knowledge base accumulates these extensions. There's a strategy in place to prevent it from becoming a tangled mess:

- **Graph Modularization:** The knowledge base might consist of many smaller graph files rather than one gigantic file. For instance, each persona could have its own graph file, each data source its own graph, and then a set of mapping files link them. This modular approach makes it easier to evolve parts of the knowledge without risking the integrity of everything at once.
- **Graph Pruning/Cleaning:** Because we have version history, we can identify when certain nodes or subgraphs become obsolete (e.g., a concept that was introduced but later deemed irrelevant). Those can be deprecated in a new version, knowing that the history is still there if needed.

Automated tests (data tests) can help identify anomalies or unused parts of the graph over time, which maintainers can then clean up.

- **Scalability:** The use of cloud-native techniques – like storing JSON in S3, using AWS Lambda or similar to run MGraph-DB functions on demand – means the solution can scale horizontally. Need to process 100 new documents? Spin up 100 short-lived graph-building functions in parallel, each outputting a graph fragment. Later, combine those fragments. The cost is linear with usage; there's no monolithic database engine choking on peak loads or idle during quiet times. This serverless ethos, enabled by MGraph-DB's lightweight nature, **keeps costs efficient and performance high**. Traditional graph databases often require a permanently running cluster, whereas here we can allocate resources dynamically.

Role of the Knowledge Base:
The knowledge base is the **single source of truth** for all derived knowledge. It's not just a cache or temporary memory – it's a persistent, queryable repository that applications and users can draw upon. For example, if a question arises ("Has concept X ever been linked to concept Y in our analyses?"), one can query the knowledge base (via an MGraph-DB query or even an NLP query if a layer is added) to find the answer. The knowledge base also serves to **feed the LLM context** for future operations. Instead of prompting the LLM from scratch each time, we can retrieve relevant portions of the knowledge base (graph subtrees) and provide them as context to the LLM for more informed output (this is similar to Retrieval-Augmented Generation).

Moreover, the knowledge base is designed to be **human-friendly** despite being machine-readable. Stakeholders can browse it – either by loading it into graph visualization tools or even by reading the structured JSON – to understand what knowledge has been captured. This transparency fosters trust in the system's outputs because one can always drill down from an answer back to the source graphs and ultimately to the original data (provenance). In essence, the knowledge base built on MGraph-DB is the backbone of an **explainable AI system**: it holds all the intermediate knowledge that explains why the AI makes certain recommendations or conclusions.

In summary, graphs from the LLM are stored as JSON-based knowledge graphs in a version-controlled repository, managed through deterministic pipelines and MGraph-DB's in-memory capabilities. The knowledge base acts as a growing, evolving library of interconnected data – curated by LLMs and humans – which can be efficiently updated, queried, and trusted over time. This approach ensures that as the system learns and accumulates information, it remains **scalable, cost-efficient, and auditable**, in stark contrast to treating the LLM as a one-off black box whose outputs vanish after use. Here, nothing is lost – every piece of inferred knowledge is captured and can contribute to future reasoning.

Q5: What is meant by the concept of an "ontology of ontologies"?

The term "ontology of ontologies" refers to a meta-level organization of knowledge – essentially a framework that manages multiple ontologies and the relationships between them. In Dinis Cruz's strategy, this is a core principle to achieve scalability and flexibility in knowledge representation. Instead of enforcing one giant, "one-size-fits-all" ontology for everything (which is often untenable in practice), the idea is to allow many *individual ontologies* to coexist and then create an overarching structure that links them together.

In practice, an "ontology of ontologies" means:

- **Multiple Ontologies for Different Contexts:** Each team, domain, or context can maintain its own ontology (its own set of entity types, relationships, and taxonomies) that makes sense for them. For example, a Security team might have an ontology centered around "Threats, Vulnerabilities, Assets, Controls", while a Business team has one around "Departments, Processes, Objectives, KPIs". Traditionally, one might attempt to merge these into a single enterprise ontology – a painful and often rigid process. Instead, we **embrace their differences**. Each ontology is like a subgraph in our overall knowledge graph, largely self-contained and authored by those who know that domain best.

- **Graphs of Graphs (Meta-Graph):** The "ontology of ontologies" comes into play as a *meta-graph* that connects these sub-ontologies. Dinis sometimes refers to this as **G^3 (Graphs of Graphs of Graphs)** – it sounds abstract, but it boils down to introducing nodes that represent entire ontologies or key concepts from one ontology and linking them to nodes from another ontology. If we continue the above example, the meta-graph might link "Asset" (from Security ontology) to "IT System" (from IT department ontology) to "Revenue Application" (from Business ontology) if, say, those all refer in part to the same actual system or concept in different language. In effect, the meta-ontology contains mappings or alignments like *Asset ≈ IT System*, *IT System is a type of Application*, etc. This is an ontology about other ontologies – it describes how the different vocabularies relate at a high level.

- **Unified View with Autonomy Preserved:** The benefit of an ontology of ontologies is that it gives a **unified view** without forcing uniformity. Each team can continue using the terms and structure that work for them (ensuring adoption and accuracy), while the organization still gains a way to translate and interoperate. It's very analogous to microservices architecture in software: each service (ontology) has its own schema and API, and the meta-ontology is like the API gateway or contract that allows them to communicate. For instance, a query from a high level – "list all assets with high risk in the company" – can be answered by traversing the meta-graph: it knows that "assets" in the Security graph connect to "systems" in IT's graph which connect to records in the Business graph that carry risk information. Without an ontology-of-ontologies layer, you'd have isolated silos or you'd attempt a one-big-ontology which breaks the specialized needs of teams.

- **Human-in-the-Loop Mapping:** Establishing and maintaining the ontology of ontologies is a collaborative process. LLMs can assist by suggesting mappings (they might observe, for example, that the Finance team's concept of "Client" is essentially the same as the Sales team's "Customer" and propose a link). However, finalizing that link might involve human agreement. The knowledge base (with MGraph-DB) is used to store these cross-ontology links as first-class data. Over time, this meta-level graph evolves as new ontologies are added or as domains change. If a team renames a concept or splits it into two, the meta-ontology is updated accordingly. Because everything is version-controlled, such changes are managed with clarity – you can see when a mapping was introduced or altered.

MGraph-DB's Role:

MGraph-DB is particularly well-suited for an ontology of ontologies approach. Its file-based, modular nature means you can literally have one file per ontology and another file capturing the mappings. This modular storage mirrors the logical separation of domains. Traditional single-store graph databases struggle here: they often encourage throwing all data into one schema, which can get messy with overlapping terminologies. In our approach, since MGraph-DB is stateless and JSON-based, combining data from two ontologies is as simple as loading two JSON graph files into memory together. We don't have to force their schemas to merge; we can operate on both and create a third structure (mappings) that references elements of each. Performance-wise, MGraph-DB can handle these multi-graph traversals in memory with type safety, and then we persist the combined insights if needed.

Benefits of an Ontology of Ontologies:
- **Scalability of Knowledge Management:** As an organization grows, new domains emerge. We can plug in a new ontology (say the company acquires a biotech division, which has its own concepts of “Gene” and “Drug”). Instead of refactoring a huge ontology, we just add the new one and then map relevant parts (perhaps “Drug” maps to the existing concept “Product” in the business ontology). This scales much more gracefully and quickly.

- **Reduced Bottlenecks and More Ownership:** Each team can evolve their ontology at their own pace, without having to convene enterprise ontology meetings for every change. The ontology-of-ontologies decouples these efforts. Central governance focuses only on the mappings at the intersections (which is a more tractable problem). This is analogous to microservices allowing teams to develop independently with only API contracts to manage between them.

- **Richness and Diversity of Ontologies:** Different ontologies might model the world in different ways – one might be more fine-grained or use different class hierarchies. Allowing them to co-exist means you can capture multiple perspectives. The meta-graph then becomes a place of **translation** and **integration**. In AI terms, this could even be beneficial because the LLM can draw from the meta-graph to understand that two terms are equivalent or related, enhancing its comprehension of context when answering questions that touch multiple domains.

- **Example – “Ontology of Ontologies” in action:** Consider a concrete scenario: The concept of “Risk” means different things to different departments. Operational Risk team has an ontology detailing types of operational failures; Compliance has an ontology about regulatory risk categories; Security has one about threat risk levels. An ontology of ontologies might have a higher-level concept “Risk” that links all these. When leadership asks for a “holistic risk dashboard,” the meta-ontology allows pulling data from all three areas and aligning them. Each area didn’t have to change their definitions; we just built the bridges.

In summary, an “ontology of ontologies” is about **managing knowledge at scale through federation rather than centralization**. It is a strategy to keep graphs **organic, modular, and easily evolvable**. Dinis Cruz’s vision uses this concept to avoid the classic failure of big ontology projects (which often collapse under their own weight). With the help of MGraph-DB, this vision is implemented as a set of interconnected JSON-based graphs – each authoritative in its domain – tied together by a curated layer of mappings. This results in a semantic graph ecosystem that is both **robust and adaptable**, where explainability is preserved (one can always see which sub-graph a piece of knowledge came from) and yet the **collective intelligence** of all graphs can be utilized when needed.

Conclusion

Dinis Cruz’s approach to evolving semantic graphs and ontologies with LLMs and MGraph-DB represents a modern, agile take on knowledge management. It emphasizes **flexibility, explainability, and human collaboration** at every step. Concepts are introduced dynamically and identified through context before being consolidated – a contrast to the rigid upfront schema design of traditional OWL ontologies. Confidence and relevance scores are woven into the fabric of the graphs, providing a quantitative handle on the inherently uncertain outputs of AI and helping prioritize what the system (and its users) should pay attention to. Reasoning in this framework is driven by LLMs for intuitive leaps, backed by deterministic rules and human oversight to ensure reliability – effectively blending artificial intelligence with expert domain knowledge.

Crucially, **MGraph-DB serves as the enabling technology** that makes this all possible at scale. Its serverless, memory-first design allows knowledge graphs to be spun up, used, and torn down on demand, which aligns perfectly with cost-efficient cloud operations. By persisting data as JSON and integrating with tools like Git, it brings software engineering rigor (version control, testing, reproducibility) into the world of semantic graphs. This means the knowledge base is always auditable and improvements are trackable. Graph versioning and human-in-the-loop workflows ensure that the knowledge stays **accurate, current, and aligned with business needs** – the graphs improve over time instead of decaying.

The concept of an “ontology of ontologies” encapsulates the strategy of organizing knowledge in a federated yet connected way. It acknowledges the reality of complex enterprises: different groups speak in different ontologies, and that’s okay. By mapping these together, the organization gains a powerful “graph of graphs” – a holistic knowledge network that remains **comprehensive without being monolithic**.

In essence, the principles at play here – use LLMs where they excel, treat knowledge as code, prefer evolution over upfront perfection, and keep the human in the loop – come together to create a semantic graph ecosystem that is **robust, scalable, and business-friendly**. This approach turns knowledge graphs from static repositories into living, breathing assets: they grow with new information, adapt to new domains, and continually support decision-making through clear provenance and intelligent reasoning. For a LinkedIn-savvy audience of professionals, the message is clear: by combining cutting-edge AI with sound engineering (in the form of MGraph-DB and thoughtful ontological design), we can unlock the full potential of organizational knowledge, making it more accessible, actionable, and aligned with how people actually think and work.
