Why Latent Credibility Analysis (LCA) is a Strong Fit for Building an Application-Dependency Matrix

LCA turns conflicting dependency evidence collected from many heterogeneous sources (logs, CMDBs, scanners, code analysis, expert input) into a single, probability-weighted matrix. Its probabilistic transparency, extensibility, and empirically proven accuracy make it preferable to heuristic voting or ad-hoc trust scores, although it requires careful parameterisation and can over-fit on small or highly correlated data.

Advantages of LCA?

**Transparency and Interpretability**

**LCA provides clear probabilistic semantics** - when the algorithm says a dependency exists with 85% confidence, that number has real meaning1[2](https://ai.meta.com/research/publications/latent-credibility-analysis/). Traditional fact-finders often produce opaque scores (like "source trustworthiness = 6") that don't clearly indicate what they represent[3](https://cogcomp.seas.upenn.edu/papers/PasternackRo13.pdf).

**Modularity and Extensibility**

**LCA models can easily incorporate domain knowledge**. For application dependencies, you can include information about:

* Source reliability based on update frequency
* Temporal patterns (recent scans vs old documentation)
* Source types (automated vs manual)
* Application criticality weightings1[3](https://cogcomp.seas.upenn.edu/papers/PasternackRo13.pdf)

**Superior Accuracy**

**Experimental results demonstrate LCA models substantially outperform fact-finders** across multiple real-world datasets. In the original research, LCA variants achieved 86-90% accuracy compared to 75-87% for the best fact-finders1.

**Principled Uncertainty Quantification**

LCA provides **confidence intervals and probability distributions** rather than point estimates, crucial for dependency mapping where incorrect assumptions can cause system outages.

Disadvantages?

**Computational Complexity**

**LCA requires more computational resources** than simple fact-finders. While SimpleLCA scales linearly like traditional algorithms, more sophisticated variants (MistakeLCA, LieLCA) have quadratic complexity in the worst case1.

**Parameter Tuning Requirements**

**LCA models need careful parameter initialization and prior specification**. This requires domain expertise to set appropriate priors for source reliability and claim difficulty1.

**Data Requirements**

**LCA performs best with sufficient data** for reliable parameter estimation. In sparse dependency environments with few conflicting claims, traditional approaches might suffice1.

**Overfitting Risk**

**More complex LCA variants can overfit** with limited training data, especially when modeling per-source or per-claim difficulty parameters1.

How SimpleLCA Works?

**SimpleLCA operates as a probabilistic graphical model where each dependency claim is treated as a latent variable**1. The algorithm uses the Expectation-Maximization (EM) framework to iteratively refine both the credibility of sources and the truth of dependency claims.

The mathematical foundation rests on each source having an **honesty parameter (Hs)** representing the probability that the source provides accurate information1. For each mutual exclusion set of claims (e.g., "Application A depends on Database X" vs "Application A depends on Database Y"), SimpleLCA calculates:

P(claim is true | observations) = P(claim, observations) / Σ P(all claims, observations)

The honesty score is updated using:

Hs = (Σ expected true claims by source s) / (total claims by source s)

Think of SimpleLCA as a **smart voting system with learning**. Imagine you're trying to figure out which applications in your company actually depend on each other, but different tools and people give you conflicting information.

Traditional approaches might just count votes or weight them equally. LCA is smarter - it learns which sources tend to be more reliable over time. If a monitoring tool consistently provides accurate dependency information, the algorithm gives its future claims more weight. If manual documentation is often outdated, it receives less influence

Explain steps for Latent Credibility Analysis?

1. Model the problem
   * Each “cell” in your matrix—“Component A depends on Component B”—is a *claim* that may be **true** or **false**.
   * Each data source (scanner, log parser, SME, etc.) is a *source* that asserts one claim per mutually exclusive set (e.g., “A→B”, “A→C”, “A has no outbound dependency”).1
2. Assign latent parameters
   * **Hs** – probability a source is honest (tells the truth when it knows it).1
   * **D** or **Dm/Ds** – probability the source *knows* the truth (models task difficulty: some dependencies are harder to detect than others).1
   * Optional matrices **Pg, Pe, Pl** capture how a source guesses, mistakes, or intentionally lies when not truthful.1
3. Build the joint probability
   * LCA writes a generative story that explains how each assertion was produced.
   * Example for SimpleLCA:  
     P(bs,c∣ym,Hs)=Hsbs,ym(1−Hs∣m∣−1)(1−bs,ym)*P*(*bs*,*c*∣*ym*,*Hs*)=*Hsbs*,*ym*(∣*m*∣−11−*Hs*)(1−*bs*,*ym*)1
4. Estimate parameters and claim posteriors
   * Run Expectation–Maximisation:  
     – E-step: compute P(ym=c∣B,θt)*P*(*ym*=*c*∣*B*,*θt*) independently for every mutual-exclusion set.  
     – M-step: update θ (e.g., Hs=∑mws,mP(ym=bs)∑mws,m*Hs*=∑*mws*,*m*∑*mws*,*mP*(*ym*=*bs*)).1
   * Repeat to convergence; output posterior belief of every dependency claim plus calibrated source credibilities.
5. Derive the matrix
   * Select the highest-probability claim per cell (or keep the full probabilities for risk-aware decisions).
   * Attach confidence scores to every dependency edge, enabling “heat-map” style Dependency Structure Matrices used in tools such as IntelliJ DSM or MagicDraw.

How do you normalize the hetergogenous source data?

Adapters in SimpleLCA serve as modular transformers that ingest heterogeneous data formats and systematically convert them into a unified internal representation for lifecycle assessment.

* **Format Ingestion**  
  Each adapter handles a specific source format (e.g., CSV, JSON, XML, relational database), parsing raw files or queries into SimpleLCA’s canonical data objects.
* **Schema Mapping**  
  Adapters map source field names and structures to the SimpleLCA ontology (e.g., mapping “prod\_id” → “process.identifier”, “amount\_kg” → “flow.quantity”) to ensure consistent data semantics.
* **Unit Standardization**  
  Built-in conversion functions normalize varied measurement units (e.g., g → kg, mL → L) into the standard unit system used internally by SimpleLCA.
* **Semantic Enrichment**  
  During transformation, adapters attach metadata (e.g., data provenance, timestamps) and link flows/processes to external databases or ontologies as needed to support traceability and impact calculations.
* **Validation & Cleansing**  
  Adapters perform data quality checks—type validation, missing-value imputation, range enforcement—and flag or correct anomalies before loading into the core engine.
* **Batching & Streaming**  
  For large or continuously updated sources, adapters implement incremental loading or streaming protocols to feed data in chunks, optimizing memory usage and performance.
* **Error Handling & Logging**  
  Any parsing or conversion errors are captured in structured logs by the adapter layer, enabling users to trace issues back to original datasets for correction.

Why Not Use Off-the-Shelf LLMs for Dependency Discovery?

1. **Excessive inference cost**  
   Large language models incur hundreds of thousands of dollars in daily operating expenses for high-throughput use cases, making per-edge dependency scoring prohibitively expensive[1](https://dl.acm.org/doi/10.1145/3629527.3651436).
2. **High energy footprint**  
   Training a model like GPT-3 consumes on the order of 1.29 million kWh—and serving it year-round further amplifies carbon emissions[2](https://cacm.acm.org/blogcacm/the-energy-footprint-of-humans-and-large-language-models/).
3. **Poor interpretability**  
   LLM predictions are opaque, offering no clear provenance or confidence scores per claimed dependency, hindering governance and auditability.
4. **Scale inefficiency**  
   Applying a 175 billion-parameter model to evaluate millions of application pairs each night is computationally infeasible within typical bank SLAs.
5. **Data-privacy risks**  
   Routing sensitive call-log data through external LLM APIs risks exposing proprietary architecture details and violates strict compliance regimes.

Why Build a Custom LCA-Based Algorithm Instead?

1. **Operational cost control**  
   SimpleLCA runs in O(|claims|) time per EM iteration on commodity servers, avoiding GPU-driven inference bills.
2. **Energy efficiency**  
   Closed-form updates eliminate long GPU-hours, slashing energy usage compared to full-scale neural training[3](https://dl.acm.org/doi/10.1145/3632775.3662830).
3. **Explicit uncertainty modeling**  
   Credibility and difficulty scores quantify trust in each source and claim, enabling risk-weighted dependency graphs.
4. **Privacy preservation**  
   All computation occurs on-premises over aggregated logs—no external data sharing or appetite for proprietary APIs.
5. **Real-time scalability**  
   Online EM and nightly pruning ensure continuous, bounded-state updates for tens of thousands of applications.

Training a Custom Model: Is It Sustainable?

1. **Capital expense**  
   Even modest transformer-based models require millions in upfront GPU hardware and weeks of training time.
2. **Ongoing energy demand**  
   Fine-tuning and retraining to track evolving traffic patterns would repeatedly consume substantial kilowatt-hours.
3. **Maintenance complexity**  
   Model drift, hyperparameter tuning, and software‐stack upgrades create continuous resource drains beyond initial build.
4. **Limited incremental gain**  
   Achieving parity with a purpose-built EM algorithm demands extensive trial-and-error at great expense for marginal accuracy improvements.
5. **Governance burden**  
   Continuous validation and explainability requirements for financial institutions make black-box retrained LLMs a poor fit under supervisory scrutiny.

Credibility Threshold in SimpleLCA: Definition and Ideal Value?

**Threshold Definition**  
The credibility threshold is the cut-off probability above which a claimed dependency is treated as “true.” In practice, once the inferred probability YAB*YAB* that application A really calls B exceeds this threshold, you consider the A→B link as a genuine edge in the dependency matrix.

**Ideal Threshold in Simple Language**  
An effective threshold balances catching real connections against avoiding spurious ones. A common rule of thumb is:

* Below 0.70 – too low: you’ll include many noisy or false links and drown in false alarms.
* 0.70–0.89 – sweet spot: you capture most real dependencies while filtering out noise.
* 0.90+ – very strict: you only keep the most certain links, but risk missing some valid ones.

For large-scale bank environments, start around **0.75–0.80**. This level typically offers a good trade-off: it flags edges with reasonable confidence without overloading teams with low-reliability calls.

Adjust up or down based on how critical it is to avoid errors versus how much manual follow-up you can tolerate.

What do we do with association whose credibility is below the threshold?

The associations with lower credibility score can be fed to a manual review-feedback loop to identify False Positives and True Positives. Based on the feedback, the Credibility score of the association can be altered. In the next iterations it will consider the manually reviewed False Positive associations as a credible dependency.