ChatGPT

LETS: A Deterministic and Debuggable Data Pipeline Architecture

Authors: Dinis Cruz & ChatGPT Deep Research

Abstract

Modern data pipelines often struggle with complexity, opacity, and brittleness. In response, we introduce **LETS (Load, Extract, Transform, Save)** – a deliberate evolution of the traditional ETL/ELT paradigm designed for determinism and debuggability. LETS pipelines emphasize modular stages with built-in serialization and traceability at each step. We demonstrate LETS in practice through real-world projects (MyFeeds.ai, a Memory-FS approach, and The Cyber Boardroom) where intermediate data states are persisted as versioned files, enabling reproducibility, provenance tracking, and rich semantic enrichment. We analyze common failure patterns of legacy ETL/ELT workflows – from "black-box" transformations with no visibility to lack of version control – and show how LETS overcomes these issues. By treating file systems or cloud object stores as first-class databases and leveraging type-safe data models, LETS pipelines achieve strong CI/CD integration, **determinism**, and end-to-end transparency. This white paper formalizes the core principles underpinning LETS (Ephemerality, Traceability, Determinism, Modularity, and "Minimum Viable Propagation") and illustrates how they foster robust, explainable data workflows suitable for AI-driven applications and beyond.

Introduction

Organizations have long relied on ETL (Extract, Transform, Load) pipelines to integrate and process data [1]. In a typical ETL workflow, data is **Extracted** from source systems, **Transformed** into a desired format, and **Loaded** into a target system (e.g. a data warehouse) [2]. More recently, ELT (Extract, Load, Transform) has gained popularity – raw data is first loaded into a storage system, and transformations are applied afterwards to enable flexible, schema-on-read analytics. While ETL/ELT pipelines are effective for moving and shaping large volumes of data, they often suffer from critical shortcomings.

Complexity & "Black-Box" Design: Data pipelines can become complex entanglements of tasks that are hard to understand or modify. In many ETL systems (e.g. legacy script-based or GUI-based pipelines), data flows through multiple transformations without clear visibility into intermediate results. These pipelines behave as *black boxes*, where an error in the middle of a pipeline can be difficult to diagnose [3]. Without proper logging or breakpoints, engineers have no insight into what each step received or produced, making debugging *"challenging when something breaks or produces unexpected results."* [3] The lack of observability means that issues (e.g. a subtle data format difference, a dropped record, or a failed join) often surface only at the end when the final output is wrong – by then it's too late to easily trace where things went awry.

Lack of Versioning & Provenance: Traditional ETL tools typically do not store intermediate data versions [4] – they extract and transform data on the fly, loading results into a destination, but intermediate states exist only in memory or transient staging. Consequently, there is poor *provenance tracking*: it can be unclear which source records contributed to a given result, or how an item was transformed along the way. As Christian Kästner notes, *"a lack of versioning, provenance tracking, and*

reproducibility makes it difficult to provide any form of accountability" in production ML/data pipelines [5]. If a data mistake is discovered, engineers struggle to reproduce the exact state that led to the error because intermediate datasets were not retained. Without version control on data or transformation logic, it is hard to answer questions like "Which version of the code or input data produced this output?" or "How has this data point changed over time?" [6] [7]. This absence of lineage undermines trust in data products.

Poor Reproducibility & Testing: Because many pipelines incorporate nondeterministic steps (e.g. sampling, or in modern cases, machine learning model inference), they may not produce the same result twice from the same input. Combined with the lack of intermediate snapshots, this nondeterminism impedes debugging and rigorous testing [8]. Data pipeline testing often defaults to end-to-end checks on final outputs, since there are no convenient checkpoints to verify along the way [9] [10]. This coarse approach means that when tests do fail, pinpointing the root cause inside a complex pipeline is arduous. As one industry blog put it, *"data pipelines are more or less a black box"* during testing, so failures become detective work rather than straightforward unit tests [10]. All these factors slow down iteration and complicate Continuous Integration/Continuous Deployment (CI/CD) for data workflows.

In summary, classical ETL/ELT architectures tend to be monolithic and opaque: they load data into memory, perform a sequence of transformations hidden from view, and deposit final results, with little insight into the journey. Recognizing these issues, we propose **LETS (Load, Extract, Transform, Save)** – a reimagined pipeline architecture that prioritizes **determinism, traceability, and debuggability** from the ground up. LETS breaks processing into modular stages and, critically, **persists the outputs of each stage** (often to simple file storage) as first-class artifacts. By making intermediate data states durable and inspectable, LETS enables a more controlled, "white-box" pipeline where every step can be understood, tested, and reproduced.

The remainder of this paper defines the LETS architecture and its implementation details. We share concrete examples from projects co-authored by Dinis Cruz and the ChatGPT Deep Research team, including **MyFeeds.ai** (a personalized news feed generator), a **Memory-FS** (memory-plus-filesystem) graph database approach, and **The Cyber Boardroom** (a serverless GenAI-driven cybersecurity advisor). We then examine how LETS addresses common failure patterns of ETL pipelines, and we distill the fundamental principles (ephemerality, traceability, determinism, modularity, etc.) that guide LETS design. Finally, we highlight how LETS facilitates strong integration with modern development practices (from type-safe data models to CI/CD) to reliably deploy AI-powered data products.

From ETL to LETS: Load, Extract, Transform, Save

LETS stands for **Load, Extract, Transform, Save**, reflecting a four-stage pipeline pattern. It can be seen as an evolution of ELT/ETL that deliberately adds an explicit *save* step after each major phase of processing. **Figure 1** below illustrates the high-level flow of a LETS pipeline, which we detail in this section.

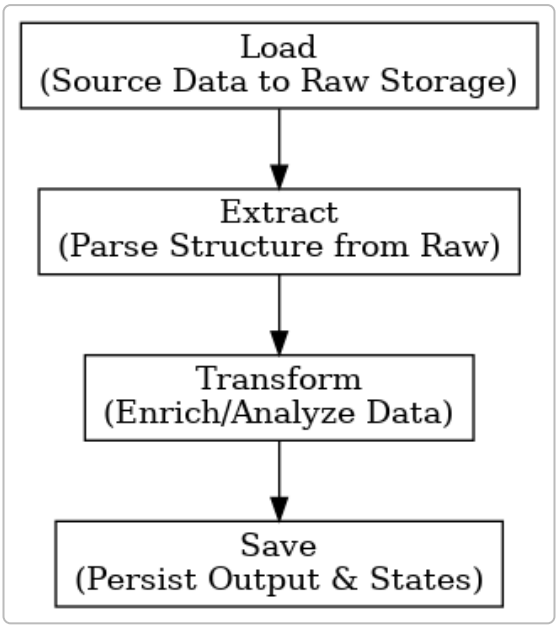

Figure 1: The LETS pipeline breaks data processing into four stages – Load, Extract, Transform, and Save – with persistent outputs at each stage. This modular design ensures each step's input and output can be inspected or replayed independently, improving traceability and debugging.

- **Load:** In the first stage, raw data is *loaded* from external sources into the pipeline's environment. This usually involves reading from an API, database, file, or web endpoint. In LETS, the key distinction is that the raw data is immediately saved to a **raw storage** layer (often a file system or object store) before any complex processing occurs. For example, the MyFeeds.ai project begins by pulling in content from RSS feeds (in the MVP, from *The Hacker News* RSS) and storing the raw feed data as a JSON file [11] [12]. By loading and capturing source data verbatim (e.g. as timestamped JSON or CSV in cloud storage), we establish an immutable baseline. This guarantees that the exact input to the pipeline is preserved, enabling reproducibility. If downstream steps produce unexpected results, engineers can always retrieve the originally loaded data and replay transformations deterministically. In essence, **Load** in LETS serves a similar role as "Extract" in ETL, but emphasizes the *persisting* of the extracted raw data in a staging area.

- **Extract:** The second stage focuses on extracting structure from the raw input. This often means parsing or converting raw data into a structured form that is easier to work with in subsequent transformations. For example, if **Load** fetched an XML or HTML document, the **Extract** step might parse it into structured JSON records. In data integration scenarios, Extract might also perform initial data cleaning or normalization (e.g. splitting a single file into multiple tables). Crucially, **Extract** in LETS is kept separate from heavy transformations – it is about *structuring the data*, not deriving insights or doing complex calculations. In MyFeeds.ai, after loading the RSS feed, an extract-like phase processes the feed entries into a more workable representation (assigning each article a unique ID, and storing each article's content in its own JSON file) [13] [14]. Each article's JSON includes fields like title, author, published date, etc., and is saved to the file system for traceability [15] [14]. By the end of **Extract**, the pipeline has a standardized, structured dataset (or set of files) derived from the raw input. This structured output is again **saved** – often as a collection of JSON files or a structured table – providing a checkpoint that can be examined. At this point, nothing algorithmically complex or model-driven has happened; we've simply prepared and serialized the data in a predictable way.

- **Transform:** In the third stage, the real data transformations take place. This is where business logic, analytics, or machine learning models are applied to the structured data obtained from the Extract stage. In a LETS pipeline, the **Transform** step can be arbitrarily complex (e.g. merging data sets, enriching with external knowledge, running ML inference) but it remains *deterministic and debuggable* because of how it's structured. Each Transform step consumes structured inputs (often from files produced in Extract) and produces **structured outputs** that are immediately saved. The use of structured outputs is a hallmark of LETS. Rather than spitting out unstructured text or just inserting into a database, each Transform stage is designed to output data in a self-describing format (like JSON following a prescribed schema). This was exemplified in MyFeeds.ai's multi-phase LLM processing pipeline – instead of doing everything in one black-box AI call, it performs a series of transformations with clear inputs/outputs [16] [17]. For instance, one transform uses an LLM to perform *"Entity & Relationship Extraction"* from an article's text, producing a JSON file capturing the article's key entities and how they relate [18]. Another transform stage takes a persona's profile and generates a *persona interest graph* (also JSON) [19]. Yet another transform compares the two graphs to map relevant overlaps, outputting a structured list of relevant entities connecting the article to the persona [20] [21]. Finally, a transform stage might generate a summary text, but even then, intermediate data (like which entities were highlighted) was preserved. Each of these Transform sub-steps in MyFeeds was implemented as a distinct API endpoint (Flow 6, Flow 7, etc. in the system) operating on data from the previous step and saving new results【35†】 [22]. By chaining multiple fine-grained transforms, the system avoids a single massive step that is hard to debug. More importantly, because each transform writes its output to a file, we gain **traceability**: one can inspect any intermediate JSON to verify the transform's correctness or to explain decisions. Structured outputs impose a kind of rigor on LLM usage as well – the LLM is prompted to fill specific fields in a JSON schema, which yields more consistent, controlled results [23] [24]. This approach was key to making MyFeeds.ai's LLM pipeline *deterministic-ish*; as Dinis Cruz notes, *"having [LLMs] fill out predefined JSON schemas at each step makes their behavior more predictable… yielding more consistent outputs and reducing variability."* [25] Any deviation (e.g. malformed JSON) can be immediately caught and handled, instead of silently corrupting the pipeline.

- **Save:** The final stage of the LETS cycle is *Save*, which in practice occurs after every stage (hence we often speak of persistent outputs throughout). However, in a conceptual sense, **Save** represents the commitment of transformed results to their ultimate destination or storage medium. In a traditional ETL pipeline, "Load" meant loading into a database/warehouse at the end. In LETS, because we have been saving intermediate files all along, the "Save" step is more about finalizing and versioning the outputs. For example, once MyFeeds.ai has generated personalized article summaries for a persona, these summaries (and their supporting data like graphs) are saved to a repository where they can be served to users. Importantly, LETS treats file systems or cloud object storage as **first-class databases** for these outputs and intermediate states. Rather than always inserting final results into a hidden SQL table, LETS often writes them to a transparent store (like Amazon S3, local disk, or a Git repository) in a structured form that developers can directly access. One pattern is to use a human-readable and diffable format (JSON, Markdown, CSV, etc.) for saved outputs so that they can be easily version-controlled. In MyFeeds.ai, every artifact generated – from raw feed to timelines to knowledge graphs – is stored on S3 with a **semantic path** and timestamp [26]. In fact, the system automatically writes each file in two locations: a `/latest/` path (for the most recent state) and a timestamped folder (to preserve a historical version of that state) [26]. For example, an intermediate graph of article timestamps might be saved to `.../latest/feed-timeline.mgraph.json` and simultaneously to `.../2025/03/26/11/feed-timeline.mgraph.json` (indicating it was created on March 26, 2025 at hour 11) [26]. This dual-save approach ensures that one can always "time-travel" to any prior intermediate result, satisfying strong provenance and versioning

requirements. It also means the pipeline is **deterministic and reproducible**: running the same Transform on the same input file will produce the same output file (content and name), and if code or input changes, a new version is simply stored at a new timestamped path rather than silently overwriting the old data. In short, **Save** in LETS cements the pipeline's outputs (at all stages) as durable, versioned data assets.

To summarize, LETS differs from classical ETL/ELT by making *every stage an explicit, serializable step*. Loading, extraction, and transformation are not just in-memory operations feeding into each other; they are distinct phases that hand off data via persisted artifacts. The pipeline becomes a sequence of stateful transitions on data, where the state is externalized (to files/objects) rather than hidden. This yields a number of immediate benefits:

- **Traceability:** Each step's output can be traced and audited. If a final result is wrong, one can walk backwards through the saved states (using IDs, timestamps, or unique keys) to locate where the anomaly first appeared. MyFeeds.ai demonstrated this by providing a provenance trail explaining *"why a particular article was recommended to a persona"* – because you can inspect the chain of JSON outputs (article entities → persona graph → matching entities) that led to that recommendation [27]. This level of traceability turns otherwise opaque AI decisions into explainable ones: *"Every recommendation can thus be explained by tracing those intermediate entities and links, turning an opaque decision into an open, auditable one."* [27]

- **Determinism & Reproducibility:** By minimizing hidden state, LETS pipelines are much easier to run in a deterministic fashion. If needed, each stage can be re-run independently on its saved input to verify it produces the same output as before (facilitating *reproducibility tests*). The use of structured schemas for LLM outputs further enforces consistency [25]. Additionally, because all data is versioned, one can recreate a past pipeline run exactly – load the exact input file version, use the same code version (which CI/CD can pin via Git), and you should get the same intermediate and final results. This reproducibility is critical not only for debugging but also for governance (e.g. reproducing a model's decision months later for audit [5]).

- **Debuggability & Testing:** LETS provides natural hook points for testing and debugging. In a CI pipeline, for example, one could run the Extract stage on a sample input and then compare the saved output JSON to an expected JSON (version-controlled as a golden file). Because each stage is deterministic and its output is materialized, unit tests can be written for every stage of the pipeline, not just final outputs [10]. Debugging is also simplified: if an error occurs during a batch run, the pipeline can halt leaving behind the last saved state, which a developer can inspect in situ. Contrast this with a monolithic ETL job where an error might cause the entire process to roll back or fail without a trace of partial results. The explicit Save points in LETS function like checkpoints in a long computation – you can resume or rerun from any checkpoint with confidence in what data you're feeding in.

- **Modularity & Extensibility:** Each stage in LETS is a self-contained module that reads input from the prior stage's output and produces new output. This modularity aligns well with microservices or serverless function architectures, where each step can be an isolated function or API. In fact, The Cyber Boardroom's architecture embraces this – it is *"driven by two key paradigms: 1) Serverless architecture…and 2) GenAI bots/agents"* [28]. In practice, this means the pipeline's stages can be deployed as independent **serverless functions** on AWS Lambda (as was done with MyFeeds.ai's FastAPI endpoints [29]). Each function's job is small and well-defined (e.g. "Convert text to knowledge graph JSON" or "Compare two graphs"), which makes reasoning about and extending the system easier. New transformations or data sources can be added by inserting another stage or branching off intermediate files, without disrupting the entire pipeline. This

also encourages **code reuse** – e.g. the persona graph construction stage could be reused across different pipelines that need to model user interests, by feeding it the appropriate input and reading its JSON output.

With the LETS stages defined, we next explore detailed examples of how this architecture is implemented in real projects, highlighting the benefits in context.

Practical Applications of LETS Architecture

Case Study 1: MyFeeds.ai – Multi-Phase LLM Pipeline with Provenance

MyFeeds.ai is a platform that generates personalized cybersecurity news feeds for different personas (like CEO, CISO, CTO, etc.) by leveraging Large Language Models and knowledge graphs [30]. It was built from the ground up using LETS principles to ensure that the AI-driven content recommendations are explainable and deterministic. We outline how MyFeeds.ai's pipeline corresponds to LETS and how that enabled key features like provenance tracking.

Loading and Extracting RSS Feeds: The pipeline begins by *loading* raw data from RSS feeds. In the first MVP, this is specifically the Hacker News RSS feed for cybersecurity updates [11]. A serverless function (exposed as the `flow-1-download-rss-feed` API) fetches the RSS XML and immediately converts it into a JSON file [12]. This JSON file (let's call it `feed-data.json`) contains all the items from the RSS feed in a structured format. Saving it serves as the raw input snapshot. Next, an *extract* stage (`flow-2-create-articles-timeline`) processes `feed-data.json` to identify which articles are new or need updating, and it creates an **article timeline** graph structure [31]. In this process, multiple output files are saved: for example, a timeline graph (`feed-timeline.mgraph.json`), a Graphviz DOT representation (`.dot`), and a rendered PNG of the timeline for visualization [32] [33]. These files are saved to S3 both in a `latest/` folder and a time-stamped folder, establishing a versioned history [26]. The timeline graph essentially acts as a way to track "what happened when" – it's a semantic index of articles by time, created using a specialized in-memory graph library (MGraph-DB) and then serialized (we'll revisit MGraph soon) [34] [35]. At this point, the raw articles are also each extracted: the system creates individual JSON files for each article (e.g. `.../articles/{article_id}/feed-article.json`) which contain the article's title, link, author, published date, etc. [36] [14]. This corresponds to the **Extract** phase – turning raw RSS into structured article objects.

Transforming with LLMs and Knowledge Graphs: Once the articles are identified and extracted, MyFeeds.ai enters a series of **Transform** stages that utilize LLMs to add semantic structure and personalization:

- **Entity & Relationship Extraction (Article Graph):** Each article's text (or summary) is fed into an LLM to extract key entities (people, organizations, technologies) and relationships among them [37]. Rather than doing this in an ad-hoc way, the system uses OpenAI's *structured output* feature with a predefined schema of Python classes [38]. In practice, Dinis defined data classes (using the OSBot-TypeSafe framework) for things like `Article_Entity` and `Relationship`, and provided these as a schema to the LLM [38] [39]. The LLM then returns a JSON (parsed into Python objects) populating these fields – e.g. listing entities like "GraphQL" or "OpenAI" and relationships like "uses" or "founded by" between entities. This output is saved as a JSON file representing the **Article Knowledge Graph** [40] [41]. By having the LLM fill a structured format, the pipeline ensures each article graph is comparable and machine-readable, rather than free-form text. As noted in the MyFeeds architecture description, *"every LLM stage outputs a structured JSON file*

rather than free-form text", which brings determinism and traceability to an otherwise probabilistic process [23].

- **Persona Graph Construction:** In parallel, for each target persona (say, the profile of a CEO interested in cloud security and DevOps), the system constructs a **Persona Knowledge Graph** using another LLM call [42]. Essentially, given a description of the persona (their role, interests), the LLM generates a graph of topics the persona cares about (for a CTO, nodes might include "microservices", "cloud infrastructure", "cybersecurity") [43] [44]. This persona graph is also output as JSON, providing a semantic fingerprint of the persona's interests [44]. In LETS terms, this is another Transform stage: input = persona description, output = structured graph JSON.

- **Relevance Mapping (Article ↔ Persona):** Next, MyFeeds.ai performs a transform to connect the article graph with the persona graph and determine which articles are relevant to which personas [45]. An LLM (or even simpler algorithm) compares the two sets of entities and finds overlaps, outputting a JSON structure that explicitly links, say, "Edge computing" in the article to "Cloud" in the persona's interests [45] [46]. Articles with sufficient overlap get a higher relevance score for that persona. This output, again saved to JSON, provides the *why* behind the recommendation – it lists which entities matched and is effectively the provenance data for the recommendation decision [45] [27]. By chaining through these steps, at this point the system has: article graphs, persona graphs, and mapping data explaining the intersection.

- **Personalized Summary Generation:** In the final transform, the system uses an LLM to generate a short summary of each relevant article, tailored to the persona [47]. The prompt for this LLM is assembled using the original article content and the highlighted relevant points (entities) for the persona [47]. The LLM then produces a summary text that emphasizes what a CEO (for example) would care about in that article. This stage outputs the summary (which might be saved as a Markdown or text file). Since the summary is derived from all the prior structured data, it is explainable – one can point to the exact reasons (via the JSON trail) why the summary mentions certain points [27].

Throughout these transforms, MyFeeds.ai followed LETS religiously by saving each intermediate. In fact, the system was implemented as a series of FastAPI endpoints (Flow 3, Flow 4, … Flow 11, etc.) each corresponding to a step that reads some files and writes others【35†】. For instance, `flow-6-article-step-3-llm-text-to-entities` would read an article's markdown or text and output the entities JSON【35†】. `flow-7-article-step-4-create-text-entities-graphs` would take that JSON and create a graph structure (perhaps using MGraph-DB)【35†】. At each juncture, the state of the article's processing is indicated by a "step number" and the presence of particular files in S3 [13] [48]. This design made the pipeline *highly debuggable*: if something failed at step 5 for article X, the engineer could retrieve article X's JSON from step 4 and investigate. It also made it *resilient*: since each step is idempotent on its input file, the system can retry or resume failed steps without starting over from scratch.

Provenance and Explainability: By the end, for each persona, MyFeeds.ai could not only deliver a set of recommended articles with summaries, but also a full provenance for each recommendation. If a user asks, "Why am I seeing this article?", the system can explain: *"Because the article mentions GraphQL and your profile lists GraphQL as an interest"* [27]. This answer is backed by the actual JSON evidence stored at each stage, making it credible and auditable. Such explainability is rarely possible in a one-shot pipeline or end-to-end ML model. As the LinkedIn article on MyFeeds.ai concludes, *"by making each step transparent and capturing its output as data, the system makes it possible to understand and trust why a specific article is shown to a given persona"* [49].

In summary, MyFeeds.ai showcases the LETS approach in an AI context: it uses **modular LLM transformations** with **type-safe outputs** to ensure each stage is check-pointed and interpretable. The result is a complex personalized content feed that nonetheless behaves deterministically, with **every intermediate reasoning step materialized for debugging**. This starkly contrasts with naïve approaches where one might prompt an LLM with all data at once and get an answer – such a one-shot method, as Dinis noted, had *"a large number of very important problems"* including lack of explainability [50]. LETS solved those by splitting the problem and saving each split's result.

Case Study 2: Memory-FS and MGraph-AI – Treating File Systems as Databases

One of the core tenets of LETS is treating the *file system or object store as a first-class database*. This means using files (often in JSON or other structured formats) as the primary means of storing and querying intermediate data, instead of always relying on an external database or only in-memory objects. The rationale is that file systems (especially cloud stores like S3) offer cheap, scalable, and easily versioned storage, and modern computing environments (like serverless) can often interact with them as conveniently as with a database.

To illustrate this concept, consider the development of **MGraph-AI**, an open-source "Memory-First Graph Database" designed by Dinis Cruz to support GenAI and serverless apps [51]. MGraph-AI was built to overcome limitations of traditional graph databases in serverless contexts, such as their weight and complexity in deployment [52]. The key design decision was that MGraph would keep graphs **in-memory** during computation for speed, but persist everything to the **file system (as JSON)** for durability [53]. Essentially, every update to the graph is serialized to a JSON file (or set of files) on disk, making the file system the source of truth. This aligns perfectly with LETS: memory provides fast transient processing (ephemeral compute), and the file system provides a stable, versionable state (persistent data). The result is getting *"the performance of an in-memory database with the reliability of persistent storage."* [53]

In a LETS pipeline, using a memory-file system hybrid (Memory-FS) approach means that each stage can operate in memory for efficiency (especially important for large graph manipulations or heavy compute) but must *save its results to disk* before handing off to the next stage. MGraph-AI demonstrates this by offering a robust, type-safe API where any graph operation (e.g. adding a node) modifies the in-memory graph and also triggers a JSON serialization (which can be diffed or version controlled) [54] [55]. This is similar to how a data pipeline transform would produce a new JSON output. The advantage is clear: if the system crashes or needs to scale down (as in serverless), no data is lost – the state is already saved; if we want to inspect what the graph looked like at a point in time, we have the file snapshots. In fact, one of the listed use cases for MGraph-AI was *"applications requiring the need to use JSON and Cloud file systems as the main data store"*, as well as *"version control and easy diffs of Graph Data"* [56]. These are exactly LETS philosophies. Traditional ETL pipelines often ignore the file system except maybe for initial ingestion, whereas LETS embraces it throughout.

Memory-FS in Practice – Cyber Boardroom: The Cyber Boardroom, which provides GenAI cyber advisory (Athena bot) to board members, employs serverless functions and likely a graph-based backend for knowledge (since it deals with Q&A and content library) [28] [57]. We can infer that it uses a pipeline to ingest and transform information (e.g. ingesting security news, company data, regulatory text) into a knowledge base that Athena can query. Given Dinis's work, it's plausible that Cyber Boardroom uses MGraph or a similar memory-FS approach for its knowledge graph. For instance, when new guidance or articles are added to the library, a pipeline might: Load the content from an external source, Extract important facts/entities, Transform into an enriched graph or embedding index, and Save it to a persistent store (which could be JSON in S3, or a Git versioned knowledge repository). The "serverless" paradigm means each step runs in isolation (ephemerality), so passing state via persistent

storage (S3 or database) is necessary – a perfect fit for LETS where each step reads/writes state. By using a file-oriented data store, The Cyber Boardroom can achieve virtually zero-cost idle persistence (files in S3 cost very little when not accessed, unlike an always-on database) and scale out its GenAI bots without centralized bottlenecks. Athena, the GenAI advisor, can be stateless itself – each query it handles can fetch whatever data it needs from the knowledge files produced by upstream pipelines. This separation of concerns (data preparation pipeline vs. query-time usage) again highlights modularity.

Type Safety and Schema Enforcement: A significant challenge in pipeline architectures is ensuring that data passed between stages adheres to expected structures. Type mismatches or schema drift can cause runtime errors or, worse, silent data corruption. LETS mitigates this by favoring explicit schemas and type-safe models for serialization. In the MyFeeds example, the use of OSBot-Utils **Type_Safe** classes to define the schema for LLM outputs is a prime case [38]. Dinis's OSBot TypeSafe framework (part of the OSBot-Utils library) lets developers define Python classes that mirror the JSON structure, so that when an LLM or any transform produces JSON, it can be directly parsed into these classes (validating types and required fields). In a sense, this is analogous to how protocol buffers or Avro schemas might be used in a data pipeline, but here applied to JSON and even LLM interactions. The benefit is that each Save output is not just a blob of JSON; it's an instance of a known data model. This makes downstream code safer (developers can use IDE auto-completion on objects rather than dictionaries) and integrates with CI tests (one can deserialize a saved file and verify its attributes). For example, if the Persona graph schema expects a list of `Interest` objects and the LLM returned something else, the deserialization would fail fast, flagging the issue.

In MGraph-AI, type-safe design is also evident. The library provides typed classes for graph elements (nodes, edges) and ensures that only those go in/out. It uses Pydantic or similar under the hood to ease JSON (de)serialization. The **combination of type-safe models and file serialization** means a LETS pipeline's contract between stages is well-defined. This is in stark contrast to many ETL scripts where one stage might output, say, a CSV with a certain column order and the next stage just assumes that order – a change in one can break the other without warning. In LETS, if one stage's output schema changes, the next stage's data class would likely not match, causing an error that can be caught during development or testing.

Minimum Viable Propagation: A subtle principle observed in LETS development (especially with memory-FS and versioning) is what we term *Minimum Viable Propagation*. This principle dictates that each pipeline run or each data update should propagate *only the necessary incremental changes* through the pipeline, rather than reprocessing everything wholesale. Because LETS stores intermediate states and uses timestamp/versioning, it can detect what's new or what needs updating. For instance, if only one new article appears in the RSS feed, MyFeeds.ai doesn't need to rebuild graphs for all previous articles – it can just process that one article through the stages and merge it with existing outputs (thanks to the graph data store). The timeline and file naming convention (`2025/03/26/11/...`) inherently support incremental processing: each hour's run produces a new folder of results without altering past folders [26]. In a similar vein, MGraph's approach to versioning each graph change allows one to propagate just that change – e.g. if a new node is added, it creates a new JSON diff – rather than recompute the entire graph from scratch. Minimum Viable Propagation is about efficiency and simplicity: do the least amount of work needed to move data from raw input to final output, and avoid touching unchanged data. This reduces the surface for errors and makes the pipeline's behavior easier to predict. It also improves CI/CD, since a small change in input or code ideally only triggers a small set of outputs to change (which can be verified via tests or diffs).

In summary, the Memory-FS approach and tools like MGraph-AI illustrate how LETS leverages **ephemeral compute with persistent storage** to get the best of both worlds. We treat files not as mere

byproducts but as a **living database** of our pipeline's state, queryable and version-controlled. This makes the architecture cloud-native (object stores are abundant and cheap) and developer-friendly (files can be inspected with simple tools, diffed with Git, etc.). By using type-safe models for these files, we introduce compile-time or test-time checks on the pipeline's integrity, providing early warning of any incompatibilities.

Case Study 3: The Cyber Boardroom – GenAI Pipeline for Decision Support

The Cyber Boardroom is a unique application of LETS principles in the domain of cybersecurity governance. It delivers an AI-powered advisor (Athena) to board-level executives, helping them stay informed and make decisions on cyber risk [58] [59]. Under the hood, The Cyber Boardroom likely integrates multiple data sources – threat intel feeds, best practice guides, regulatory texts, Q&A pairs – and uses generative AI to present insights conversationally. We examine how such a system benefits from a LETS architecture.

Serverless GenAI Bots with Pipeline Backend: Athena, the AI advisor in The Cyber Boardroom, is described as a *"python-based serverless GenAI cyber security advisor"* [60]. "Serverless" implies that whenever a user interacts (asks a question), a cloud function spins up, loads the necessary data/model, computes a response, and then terminates – there's no persistent process holding context between sessions. This statelessness at query time means any context Athena has (such as the knowledge of the company's policies or recent cyber news) must be loaded from a storage. This is where a LETS-structured pipeline comes in: upstream, there are likely scheduled pipelines that **Load** relevant data (e.g. daily news, new laws), **Extract** and normalize it (e.g. parse news articles or PDFs into text snippets, tag them, index them), **Transform** it into a knowledge store (perhaps vector embeddings for semantic search, or a graph linking topics), and **Save** this enriched knowledge base.

For example, one could imagine a pipeline that daily ingests security news articles similarly to MyFeeds (load RSS, extract text, use an LLM to summarize key points or classify the risk type, save those as JSON files in a "news" bucket). Another pipeline might ingest official cybersecurity frameworks or standards (NIST, ISO) and break them into pieces, tagging which topics they cover. All these pieces of information are then saved in a structured form that Athena can query. By using LETS, if Athena gives an answer, it can also provide citations or origins of its information because the pipeline stored where each piece came from (provenance). This is crucial in an advisory context – board members will want to know the source of AI-provided advice. Because of LETS's traceability, Athena could reference *"Source: NIST CSF section 3.2"* or *"as reported by The Hacker News on 2025-03-24"*, with links, drawn directly from the pipeline outputs.

Furthermore, the **determinism** of the pipeline ensures that Athena's knowledge is consistent and up-to-date. If the pipeline is rerun (say new data arrives or a correction is needed), it produces a new versioned knowledge file. Athena's responses can then be deterministically improved by pointing to the new data. Strong CI/CD integration is apparent here: any changes to the pipeline or the data can trigger tests (e.g. verifying that Athena's answer to a sample question hasn't regressed). If the answer should change (because new data came in), that is expected and can be reviewed by experts before deployment. Essentially, LETS enables a **continuous delivery of knowledge** to the AI, with validation steps in between.

Graph and Semantic Enrichment: The Cyber Boardroom likely uses semantic graphs extensively – perhaps mapping how various risks, mitigations, regulations, and incidents relate. This is in line with Dinis's broader interest in knowledge graphs for personalization [61] [62]. By using a LETS pipeline to build and update these graphs, the system can maintain a robust semantic context for Athena. For instance, if a new vulnerability in cloud infrastructure is reported, a pipeline might update a node in the

graph (“Cloud Infrastructure”) with a new edge to a “Vulnerability” node, and attach details. All of this can be done by an LLM transform plus some logic, and saved. Athena’s next advice about cloud infrastructure security will then be aware of that new vulnerability. Without LETS, incorporating such dynamic updates into an AI assistant would be far more error-prone – one would possibly fine-tune a model or keep long prompt contexts, which are less transparent. LETS provides a **clear separation**: the pipeline handles knowledge acquisition and structuring (with full audit trail), while the GenAI bot (Athena) focuses on language interaction, pulling from that structured knowledge.

CI/CD for AI with LETS: One notable aspect is how well LETS aligns with rigorous engineering practices even in AI systems. By maintaining all intermediate artifacts, one can include them in version control (or at least checksums of them) and detect changes. A CI job could, for example, run the entire pipeline on a fixed small input set and ensure that outputs match expected files (to catch any accidental change in logic). It could also run a batch of Q&A through Athena (with a fixed seed for the LLM if possible) and compare answers to expected outputs or at least check that citations are present, JSON formats are correct, etc. Because LETS encourages **deterministic output formats**, such tests become meaningful. This is a big improvement over naive AI deployments where outputs can vary run to run; by constraining the AI with structured prompts and capturing its output as JSON, we make it testable [25].

Finally, the use of serverless (ephemeral) compute in The Cyber Boardroom demonstrates Ephemerality in LETS: no process holds state in memory between runs; everything is ephemeral except what’s in storage. This enforced discipline means all state *must* be saved (otherwise it’s lost when function ends). While this might seem like a constraint, it’s actually a blessing for determinism and traceability – nothing is hiding in a runtime memory that could later diverge from recorded outputs. This principle of Ephemerality (in compute) combined with Persistence (in storage) is exactly what LETS formalizes.

Discussion: Advantages of LETS over Traditional Pipelines

Having examined LETS conceptually and in practice, we can directly address how it solves the earlier-mentioned limitations of traditional ETL/ELT pipelines:

- **Transparency vs. Black-Box**: LETS turns a pipeline into a *transparent* sequence of state transformations. Each state is recorded, so the pipeline is no longer a black-box but a clear box. As one data engineering article noted, observability can prevent pipelines from being black boxes and make errors visible early [63]. LETS provides that observability inherently – one can open the intermediate files and *see* what each step did. The difference is akin to having debug-level logging of every step’s input/output, except it’s actual data files, not just logs.

- **Built-in Data Lineage**: Because data is saved with semantic filenames and often nested in timestamp/version folders, lineage is built in. We know exactly which input file led to which output file, by virtue of naming or an ID system. In contrast, many ETL systems require separate data lineage tracking solutions on top (to trace records through pipelines). LETS *is* its own lineage tracker: the folder structure or file references serve as a direct acyclic graph of lineage. For example, the `article_id` in MyFeeds.ai’s filenames connects the raw feed entry to all subsequent files for that article [15]. In more general terms, one could design a LETS pipeline such that each output file contains metadata about its source (e.g., including the source file path or ID within it). This satisfies the provenance requirement highlighted by Kaestner for accountable ML systems [64] [65].

- **Versioning and Reprocessing**: Traditional pipelines often require manual reprocessing or backfills when data changes, and without versioning this can be risky. LETS encourages always

writing new outputs rather than overwriting (or if overwriting "latest", still keeping a historical copy) [26]. This means you can re-run a pipeline at any time and not interfere with past results. New results just appear in a new dated partition, for instance. This approach is common in modern data lake practices (append-only, timestamped partitions) but LETS applies it even to intermediate steps, not just raw logs. The benefit is quicker recovery from issues: if a bug is found in Transform stage 3, you can fix it and re-run stage 3 on past data to produce corrected outputs (with a new version), without touching the raw load or earlier stages. The versioned files also enable *time travel* in analysis – you can compare yesterday's transform output to today's and see what changed (facilitating debugging and incremental updates).

- **Deterministic Testing**: Each LETS stage can be unit tested with synthetic input files to ensure it produces the expected output file. This was very hard in monolithic ETLs since they were usually tested end-to-end (if at all) [10]. Determinism is improved by reducing randomness; where randomness is needed (e.g. shuffling or sampling), one can inject a seed or record the random choices to the output. The structured outputs allow automatic validation (e.g., a JSON schema validator can run on each output to catch malformations immediately) [25].

- **Modularity and Team Productivity**: Different teams or team members can own different stages of a LETS pipeline without stepping on each other's toes. Because the contract between stages is a file schema, one could even implement one stage in a different language or framework as long as it reads/writes the agreed format. This decoupling is harder in tightly integrated ETL jobs. The modularity also means improvements in one stage (say a better LLM prompt for extraction) do not necessitate rewriting the entire pipeline – they just produce better JSON outputs, which the next stages will consume unchanged format-wise. In effect, LETS pipelines are *pipeline-as-code* in a very literal sense: each stage is like a function, the files are function arguments/returns, and you can refactor or optimize one function independently. This aligns with best practices like those Palantir suggests for high-quality data pipelines, emphasizing tracking changes in both code and data across pipeline steps [66] [67].

- **Minimum Viable Propagation (Efficiency)**: By propagating only necessary changes and using small batch sizes (even batch size of 1, as each new data item flows through individually), LETS can reduce latency and resource usage. Many classic ETLs would reload an entire day's data even if only one record changed. In LETS, if using an event-driven or incremental approach, one new record = one new file goes through. This makes near-real-time pipelines easier to implement and maintain, addressing the latency issues of ETL where insights come only after hours or days [68]. With LETS, one can design streaming pipelines where "Load" writes a small file for each event and triggers downstream stages event-by-event, yet still keep the benefits of traceable, file-based steps.

Of course, LETS is not a silver bullet. The trade-off for all this verbosity (many files, many steps) is that pipelines can initially seem more complex to set up than a quick script that does everything in one go. There is an overhead in writing data to disk frequently and managing many small files. However, tools and frameworks can help (e.g., using an object store with high I/O throughput, or using batch writes if performance demands it). Also, storage is cheap – the ability to debug and avoid errors often outweighs the cost of some extra GBs of intermediate data. Indeed, many data engineering teams have independently discovered that storing intermediate results in a data lake and making pipelines idempotent is a best practice for reliability [69] [70]. LETS formalizes that pattern and extends it beyond just raw vs final data to *every logical stage* of processing.

In scenarios with strict latency requirements (e.g. high-frequency trading data), writing to disk might seem too slow. In such cases, a hybrid approach can be used: keep an in-memory pipeline for speed but

periodically dump state to disk for traceability, or leverage fast in-memory file systems (like tmpfs or memfs) that can snapshot to disk. The **Memory-FS** concept we discussed is essentially that – hold working data in memory for speed, but treat the file system as the commit log so you never lose the plot. If performance allows, doing it every step is ideal; if not, one might combine some steps before a Save. The granularity is adjustable, but the principle of determinism remains.

LETS Principles and Best Practices

From the above exploration, we can extract a set of guiding principles that define the LETS philosophy. These serve as best practices for anyone looking to implement a LETS-style pipeline architecture:

- **Ephemerality of Compute, Persistence of Data:** Design pipeline tasks to be stateless and ephemeral (they can start, run, and terminate without retaining information internally). All state that needs to persist should be externalized to durable storage between tasks. This principle often leads to using serverless or short-lived containers for execution, and cloud storage or file systems for state. The Cyber Boardroom example (serverless GenAI functions with a persistent knowledge base) embodies this [28]. Ephemerality forces discipline in saving state and makes scaling easier (any instance can pick up a task since nothing is stuck on a single server). It also naturally aligns with *Infrastructure as Code* and CI/CD – you deploy code that runs functions on demand, rather than maintaining stateful services.

- **Traceability (Provenance & Lineage):** Every piece of data output should be traceable back to its inputs and the transformation applied. This means embedding identifiers or references in outputs that link to source data, and logging transformation metadata. In practice, this could be as simple as including a source file name or ID in the output JSON, or storing all files in an organized hierarchy by date, data source, etc. The goal is that you can pick any data record or file and determine "where did this come from?" [27]. Coupled with version control, traceability also implies auditability: one can answer who/what/when regarding any data modification [64]. Embrace tools or formats that support lineage – for instance, some modern data formats like Parquet allow adding provenance metadata. In LETS, however, even plain JSON in a well-named file can suffice as a lineage marker (human-readable and tool-readable).

- **Determinism & Reproducibility:** Strive to make each pipeline step as deterministic as possible. Where nondeterministic elements exist (e.g. random sampling, concurrency issues, or LLM temperature), either control them (fix random seeds, use temperature 0 for LLMs when feasible) or capture their results such that replays use the same data. Consistency builds trust that the pipeline will behave the same way in test and production and that issues can be reproduced for debugging [71] [8]. Reproducibility also means documenting or pinning the versions of code and dependencies used for each pipeline run (tools like Docker and poetry/pip requirements help here). A LETS best practice is to include a "pipeline run manifest" file with each batch of outputs, recording the git commit hash of the code, the timestamp, and any config parameters. This makes each set of outputs self-describing and repeatable.

- **Modularity & Single Responsibility:** Each stage of LETS should have a clear, single responsibility – load data from one source, parse a format, enrich with one kind of knowledge, etc. This modularity makes the system easier to extend and maintain. If one stage fails or underperforms, you can optimize or replace it without rewriting the whole pipeline. It also makes parallel development feasible: one developer can work on improving the Extract stage while another focuses on the Transform logic, with the interface (the saved data schema) acting as the contract between them. Keep the interfaces between stages as simple and generic as possible (e.g., use

widely readable formats like JSON, CSV, Parquet, rather than proprietary binary dumps). This ensures that even if you later swap out an implementation (say Python for Rust for performance), the rest of the pipeline doesn't need to change.

- **Type Safety & Schema Evolution:** Use typed data models to define what each stage produces. This might mean using data classes or schema definitions (JSON Schema, Protocol Buffers, Pydantic models, etc.). By validating outputs against a schema, you catch errors early and also provide downstream stages with a guarantee of data shape. When a schema needs to evolve (which it inevitably will as features are added), treat it with care – consider backward compatibility or a migration plan for old data. In LETS, because data is versioned, you might version your schemas as well. For example, you could include a schema version number in each output file's metadata, allowing stages to handle slight differences if you choose not to backfill old data. The emphasis is on not silently breaking the pipeline by changing something implicitly; any change should be explicit and traceable.

- **Minimum Viable Propagation:** Process and propagate only what you need, when you need it. If only part of the data changed, don't recompute everything – leverage your saved states to only add or update the necessary portion. This often involves designing your pipeline to be **idempotent** (re-running on the same input doesn't change the output) and **incremental**. Many LETS pipelines utilize an "update marker" or checkpoint to know where they left off. For instance, if new data arrives with an incrementing ID or timestamp, the Load stage can fetch only data after the last seen ID. Then downstream, you might merge new data with existing data (which is easier when data is stored in file partitions by date or ID range). Minimum viable propagation goes hand-in-hand with event-driven architectures; you can even trigger your pipeline on new data events and have it operate only on that event's payload. The benefit, again, is efficiency and easier isolation of errors – a bug affecting one day's data only taints that day's outputs, which you can re-run independently.

- **Observability & Monitoring:** Build monitoring into each stage – because outputs are saved, monitoring can be as simple as counting files, checking file sizes, or running data quality assertions on the outputs. If a stage suddenly produces an empty file or drastically fewer records, that should raise an alert (it could indicate an upstream source issue or a bug). Traditional pipelines might not notice a drop in records if they don't explicitly check. With LETS, since each result is materialized, it's straightforward to implement checks (e.g., compare number of entities extracted today vs 7-day average). Logging is still important – each stage can log its progress and any anomalies – but the real power is being able to interrogate the data itself for anomalies. Some teams use data observability platforms that watch for schema changes or volume changes; these could plug into a LETS file repository easily (e.g., a tool could scan the S3 bucket of outputs and apply rules). Essentially, think of your output files as tables in a data warehouse – you want to monitor their health just the same. Many incidents in data pipelines (like missing data in reports) could be prevented by early detection at the pipeline stage where the data went missing [63] [72] .

- **Documentation and Discoverability:** Finally, a principle often overlooked: document what each stage does and what each output contains. Because LETS will produce a lot of artifacts, good documentation is key to not confusing users (or future maintainers). Consider adopting a naming convention that is self-explanatory (the MyFeeds flows, for example, had descriptive names like `step-2-create-article-markdown` 【35†】). Provide a README or data dictionary for the files stored, especially if they will be analyzed by others. In some cases, teams generate an index or catalog of all outputs, maybe as a JSON or HTML summary linking to the files and describing them. This can be automated as part of the pipeline. The goal is that anyone

new can understand the pipeline's structure and data at a glance, despite the many moving pieces.

Related Work and Conclusion

The LETS architecture aligns with emerging trends in data engineering and MLOps that advocate for greater transparency and control in pipelines. Approaches like **data lineage frameworks**, **data version control (DVC)**, and **lakeFS** for versioned data lakes are all tackling similar problems [73] [74]. LETS can be seen as a conceptual framework that ties these pieces together in an actionable way for pipeline design. For instance, tools like *DVC* and *lakeFS* allow versioning data and pipeline outputs, which is exactly what LETS encourages (we manually did it with timestamped folders, but these tools provide more governance) [75] [76]. Workflow orchestrators like *Dagster* have started emphasizing software-defined assets – where each intermediate is an asset that can be monitored and saved – which resonates with LETS treating intermediate data as first-class citizens. Additionally, the rise of *event-driven pipelines* and *streaming ETL* highlights the need for incremental, minimal propagation processing, something LETS handles inherently by design.

From the perspective of AI and machine learning, LETS offers a path to **Deterministic GenAI** – a term that aptly describes MyFeeds.ai's goal of reliable LLM outputs [61] [41]. By breaking LLM tasks into smaller deterministic sub-tasks with schema-guided outputs, one can integrate AI into enterprise workflows with much more confidence. This is particularly relevant as industries like finance and healthcare look to use AI but must satisfy audit and compliance requirements. A black-box AI pipeline would not pass muster, but a LETS pipeline, where each AI decision is logged and explainable, stands a far better chance. In fact, one could argue LETS brings the spirit of traditional software engineering (modularity, testing, clear interfaces) to the otherwise less governable world of AI engineering.

In conclusion, LETS (Load, Extract, Transform, Save) is more than just a rearrangement of ETL; it is a rethinking of pipeline architecture for an era that demands **modularity, observability, and trust** in data systems. Co-authored by human expertise and AI (as exemplified by this collaboration between Dinis Cruz and ChatGPT's deep research assistance), the LETS approach embodies a fusion of solid engineering principles with cutting-edge AI capabilities. By learning from the limitations of past pipelines and leveraging modern tools, LETS pipelines are deterministic where possible, and debuggable by design.

LETS has been validated in projects like MyFeeds.ai – which delivered personalised content with full provenance [49] – and in the Cyber Boardroom – which is pioneering AI advisory in a transparent, serverless manner [28] [57]. These implementations show that we don't have to trade off **speed for safety** or **innovation for integrity**: with LETS, we can have both. Data engineers, developers, and data scientists can adopt these practices to build pipelines that not only feed data into models or dashboards, but do so in a way that every step can be trusted, verified, and improved continuously.

As data ecosystems grow ever more complex, adopting a deterministic and debuggable data pipeline architecture is a strategic choice. LETS provides a blueprint for that choice – an architecture that is ready to handle the demands of AI-driven applications and stringent governance alike. We invite the community to explore LETS, contribute ideas (perhaps new tools for automation of intermediate state management or schema evolution), and share their experiences. Ultimately, LETS is about making sure that as our pipelines load, extract, transform, and save the world's data, we as engineers can **Load, Explain, Trust, and Secure** those pipelines for the long run.

Sources

1. Dinis Cruz, “Building Semantic Knowledge Graphs with LLMs: Inside MyFeeds.ai's Multi-Phase Architecture” – *LinkedIn Article*, Mar 2025. [61] [62]

2. Dinis Cruz, “Establishing Provenance and Deterministic Behaviour in an LLM-Powered News Feed (MyFeeds.ai MVP)” – *LinkedIn Article*, Mar 2025. [23] [27]

3. Dinis Cruz, *MyFeeds.ai Architecture Posts – mvp.myfeeds.ai blog*, Mar 2025 (Parts 1 & 2). [13] [26]

4. Dinis Cruz, “Introducing MGraph-AI – A Memory-First Graph Database for GenAI and Serverless Apps” – *LinkedIn Article*, Jan 2025. [77] [53]

5. Sean Dougherty, “ETL limitations and ETL alternatives for marketers” – *Funnel.io Blog*, 2023. [4] [68]

6. Max Lukichev, “How to Test Data Pipelines: Approaches, Tools, and Tips” – *Telm.ai Blog*, Jan 2023. [10]

7. Christian Kästner, “Versioning, Provenance, and Reproducibility in Production Machine Learning” – *Medium*, 2023. [5] [64]

8. Palantir Technologies, “Why Data Pipeline Version Control Matters” – *Palantir Blog*, 2022. [67] [66]

9. Dinis Cruz, “The Cyber Boardroom is official now” – *LinkedIn Article*, Apr 2024. [28] [59]

10. Telmai, *ibid.*, (Observability of pipelines as black box). [3]

---

[1] [2] [4] [68] ETL limitations and ETL alternatives for marketers
https://funnel.io/blog/etl-limitations

[3] [9] [10] How to Test Data Pipelines: Approaches, Tools, and Tips
https://www.telm.ai/blog/how-to-test-data-pipelines/

[5] [6] [8] [64] [65] [71] [75] Versioning, Provenance, and Reproducibility in Production Machine Learning | by Christian Kästner | Medium
https://ckaestne.medium.com/versioning-provenance-and-reproducibility-in-production-machine-learning-355c48665005

[7] [66] [67] Why Data Pipeline Version Control Matters | Palantir | Palantir Blog
https://blog.palantir.com/data-pipeline-version-control-tracking-code-data-together-palantir-rfx-blog-series-3-4d1783d548a2?gi=409ed8ad2607

[11] [16] [17] [18] [19] [20] [21] [23] [24] [25] [27] [30] [37] [40] [41] [42] [43] [44] [45] [46] [47] [49] Establishing Provenance and Deterministic Behaviour in an LLM-Powered News Feed (first MyFeeds.ai MVP)
https://www.linkedin.com/pulse/establishing-provenance-deterministic-behaviour-llm-powered-cruz-dimhe

[12] [26] [29] [31] [32] [33] [34] [35] How I'm Building Personalised News Feeds with Semantic Graphs - Part 1
https://mvp.myfeeds.ai/publishing-a-new-personalised-set-of-posts-part-1/

[13] [14] [15] [22] [36] [48] How I'm Building Personalised News Feeds with Semantic Graphs - Part 2
https://mvp.myfeeds.ai/how-im-building-personalised-news-feeds-with-semantic-graphs-part-2/

28 57 58 59 60 The Cyber Boardroom LTD has been created, it's official now :)
https://www.linkedin.com/pulse/cyber-boardroom-ltd-has-been-created-its-official-now-dinis-cruz-oaj8e

38 39 50 Building Semantic Knowledge Graphs with LLMs: Inside MyFeeds.ai's Multi-Phase Architecture
https://mvp.myfeeds.ai/building-semantic-knowledge-graphs-with-llms-inside-myfeeds-ais-multi-phase-architecture/

51 52 53 54 55 56 77 Introducing: MGraph-AI - A Memory-First Graph Database for GenAI and Serverless Apps
https://www.linkedin.com/pulse/introducing-mgraph-ai-memory-first-graph-database-genai-dinis-cruz-wxmde

61 62 Building Semantic Knowledge Graphs with LLMs: Inside MyFeeds.ai's... | Dinis Cruz
https://www.linkedin.com/posts/diniscruz_building-semantic-knowledge-graphs-with-llms-activity-7309935484938407936-YGE-

63 Data Pipeline Observability: Best Practices and Strategies - DQLabs
https://www.dqlabs.ai/blog/data-pipeline-observability/

69 Challenges of building high performance data pipelines for big data ...
https://www.eyer.ai/blog/challenges-of-building-high-performance-data-pipelines-for-big-data-analytics/

70 ETL Data Pipelines: Key Concepts and Best Practices - Panoply Blog
https://blog.panoply.io/etl-data-pipeline

72 May 15 Outage Post-Mortem - Inside Skylight
https://blog.skylight.io/may-15-outage-postmoterm/

73 Data Versioning: Best Practices And Tools For Data Teams
https://www.montecarlodata.com/blog-data-versioning-guide/

74 Automatic Data Provenance for Your ML Pipeline - Valohai
https://valohai.com/blog/automatic-data-provenance-for-your-ml-pipeline/

76 Why Data Pipeline Version Control Matters - Palantir Blog
https://blog.palantir.com/data-pipeline-version-control-tracking-code-data-together-palantir-rfx-blog-series-3-4d1783d548a2