The Future Is Bespoke: Synthesizing One-Size-Fits-One DBMSs with LLM Coding Agents

Timo Eckmann* Matthias Jasny* Johannes Wehrstein* Carsten Binnig

* Authors with equal contribution.

Abstract

In this paper, we propose a new direction for data management: on-demand synthesis of workload-specific database systems, which we call Bespoke DBMSs. Rather than designing a single general-purpose engine to serve different workloads—i.e., one engine for all OLAP workloads or one for all OLTP workloads—we leverage the code generation capabilities of large language models (LLMs) to automatically construct bespoke database engines that are specialized for a specific workload, such as TPC-C or TPC-H. While LLMs can already generate individual code fragments, synthesizing full database systems requires new pipelines that incrementally guide LLM agents from generating isolated building blocks—such as storage formats, operators, and execution logic—to assembling complete, coherent, and executable systems. In this paper, we present a vision of how to enable the generation of Bespoke DBMSs and sketch a synthesis pipeline that, given a workload specification, generates an end-to-end database engine fully automatically. Through a case study, we demonstrate that such a synthesized DBMS can significantly outperform general-purpose engines, pointing toward one-size-fits-one DBMSs.

1 Introduction

One size does not fit all. Modern database systems are predominantly designed as general-purpose execution engines. Since the early 1980s, major vendors have pursued what Stonebraker and Çetintemel termed the *"One Size fits All"* strategy: maintaining a single code base intended to serve all database applications [1]. A single system is expected to support a wide spectrum of workloads, from highly concurrent transactional workloads to long-running analytical queries, while accommodating diverse schemas, access patterns, and isolation requirements. To this end, over the past decades, different types of database systems have been developed, focusing on specific workload classes to better support specific workload patterns.[2–10]. However, in this paper, we argue that current database systems still incur unnecessarily high overhead because they aim to be general-purpose.

The Performance-Tax is Inherent, Not Accidental. To be more precise, think of OLAP or OLTP as prominent workload classes where specific database engines per workload class exist. Applications using such an OLAP or OLTP database can still design any database schema and are designed to run any query on this schema. And this fact comes with significant overhead and unnecessary performance penalties as database engines are designed to serve any workload that can be expressed within the boundaries of the relational This causes unnecessary overhead. A simple yet non-negligible example of such overhead is that DBMSs need to interpret a schema when a query comes in. Moreover, rooted in the same problem, databases use general-purpose data structures and algorithms to store tables instead of hard-coding the schema or using application-optimal data structures and query processing algorithms. model using tables and SQL. However, this does not mean that these engines are poorly designed. Instead, each source of overhead is a reasonable response to flexibility.

Workload-Specific DBMSs to the Rescue? Consequently, the natural question is how much of this overhead can be removed when designing a database for one specific workload. A system designed for one specific workload and not a workload class can shed the abstractions that generality demands. Schema interpretation can be removed, data structures for storing data and query processing algorithms or mechanisms for transaction handling can match access patterns exactly, and unused code paths can be eliminated entirely. To make it short, the most optimal database engine is an engine that can be hard-coded toward one particular workload and remove all unnecessary artifacts that result from the demand of flexibility. However, is this realistic? First, we see already such attempts today. TigerBeetle [10] is an OLTP database that can only execute debit-credit transactions and has been shown to significantly outperform general-purpose database systems by adapting the engine, including concurrency control, to the specifics of the debit-credit workload. However, TigerBeetle has been manually designed, and the effort of building an engine for one workload has paid off in this case as the market is large enough. As such, it is questionable if bespoke engines such could also arise for other workloads?

Manual Construction Will Not Scale. Manually designing a database engine comes with a significant investment and complexity as still solutions for all DBMS components including storage management, concurrency control, crash recovery, query execution, and countless edge cases must be engineered and validated over years of production use. Consequently, even specialized database systems comprise hundreds of thousands of lines of code. As such, building a specialized engine for a particular workload, while it has the potential to improve performance significantly through specialization, requires engineering effort on a similar order as building a general-purpose engine. And in a bespoke world, where we envision one database engine per workload, this overhead would multiply by the number of workloads, which likely grows into the millions if sufficient. As a result, bespoke database engines, if engineered manually, make sense only if the market is really large, such as in the TigerBeetle case. Consequently, a vast majority of workloads must accept performance of general-purpose engines for their workload class, not because specialization would not help, but because no one can afford to build a bespoke engine for every application. For now at least.

Our Vision: Synthesis of Bespoke (One-size-fits-One) DBMSs. As shown in Figure 1, in this paper we present our vision of synthesizing *workload-specific database engines* on demand based on significant advances in AI coding agents [11–13, 13–16]. Given a workload specification — at the core, a schema and queries — our AI-rooted synthesis pipeline can produce a complete database engine specialized to exactly the queries and transactions. Crucially, because synthesis operates over the entire workload at once, it can exploit semantic relationships across queries and transactions that no per-query compiler can see, for instance, generating workload-specific execution paths and specialized data structures when one transaction produces values that another consumes. However, as we show in this paper, naively using AI coding agents to build a full system fails due to its complexity. Instead, key ingredients such as correctness and performance validation to steer generation, as well as an incremental procedure that gradually increases code complexity, are required to generate correct and fast database engines. As we show in this paper, we can thus synthesize engines even for complex workloads in the order of hours, at the cost of a few $$$, while outperforming general-purpose engines by an order of magnitude—making the idea of Bespoke Databases attractive for any workload.

Contributions and Outline. In Section 2, we present our vision of bespoke databases in more detail. In Section 3, we describe a first synthesis pipeline for OLAP engines that, given a workload specification, incrementally guides LLM agents from a correct to a highly optimized bespoke database engine. We validate this vision through a case study, synthesizing bespoke OLAP engines for TPC-H[17] in Section 4. In Section 5, we discuss opportunities and challenges for synthesizing bespoke OLTP engines. Finally, to conclude we outline the road ahead in Section 6 as this paper can only be a starting point of a Bespoke Future and conclude in Section 7.

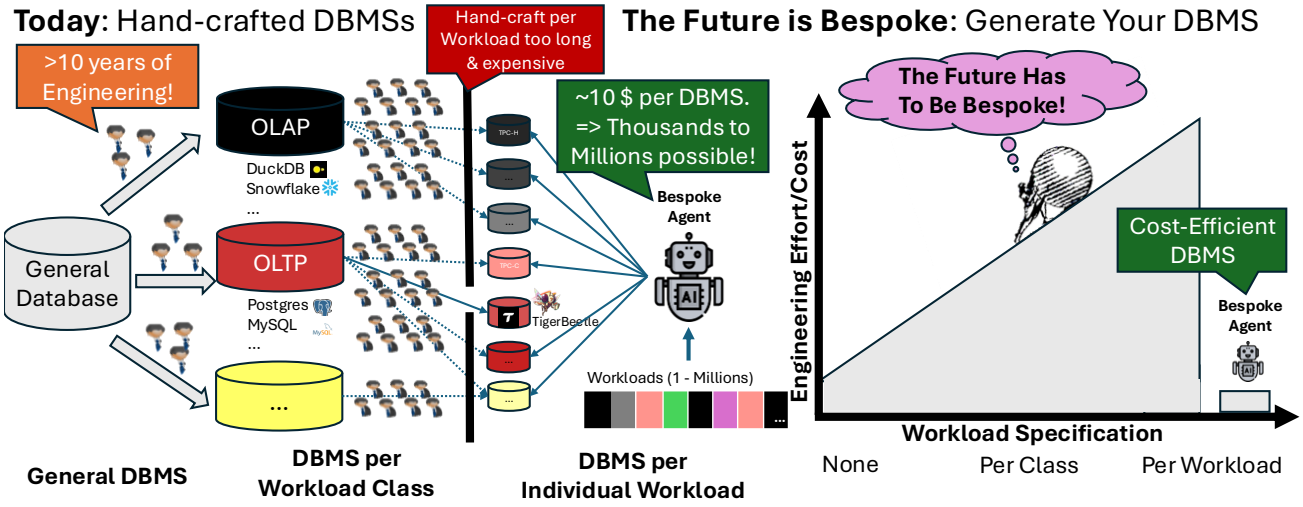

Figure 1: The Future Is Bespoke. Database systems has progressed from general-purpose systems to workload-class-specific engines (e.g., OLAP, OLTP) through over a decade of effort—but hand-crafting a DBMS per individual workload remains prohibitively expensive (only done in rare cases e.g. TigerBeetle[10]). AI-driven bespoke synthesis closes this gap, enabling the synthesis of bespoke DBMSs—one database that fits one workload—at low cost (a few $$$) and in the order of hours.

2 Our Vision: Bespoke DBMS

In the following, we detail our vision of Bespoke DBMSs and why this is not just a new way of tuning DBMSs for a given workload.

2.1 Today: Workload-Optimization as an Afterthought

General Purpose Engines Require Workloads to Adapt. In conventional database deployment, engines are first built and then users implement an application on top. However, this needs massive handholding and tuning to achieve high performance. Users therefore need to design and optimize their database schemas, design and tune queries to satisfy the optimizer, and tune various parameters to approximate acceptable performance. Moreover, specialization for a workload, if it occurs, happens through other general-purpose mechanisms: index selection, materialized views, partitioning schemes, all layered atop a general-purpose architecture that is fundamentally workload-agnostic at its core.[18–21]
Bespoke Databases Invert this Relationship. In our vision, the workload comes first. The user declares what the database must do; the system generates an engine shaped entirely by the workload declaration. Assumptions that a general-purpose engine cannot make become the foundation of a bespoke design. For example, if a workload specification declares that two transactions never conflict, concurrency control can be eliminated completely. If queries always access data in timestamp order or any other application-specific manner, storage can be organized accordingly. If the dataset fits in memory, buffer management completely disappears. Each assumption the specification permits is an assumption the generated engine can exploit.

2.2 Future: Bespoke DBMSs = One Size Fits One

Stonebraker and Çetintemel argued that "one size fits all" had failed and that we needed specialized systems for distinct workload classes [1]. The database community delivered: dedicated OLTP engines[7,

22, 23], OLAP engines[9, 24, 25], stream processors[26, 27], and graph databases[28, 29]. Yet each engine still remains general-purpose within its class. We push this to its logical conclusion: one size should fit exactly one.

The Specification Is a Contract. The input to bespoke database synthesis is a workload specification consisting of three components: (1) a *schema* describing the logical structure of the data; (2) a set of *query or transaction templates* defining the operations the engine must support; and (3) *performance objectives* such as throughput targets, latency bounds, or resource constraints. The user commits to a bounded workload; the pipeline commits to an engine optimized for exactly that workload. Critically, queries or transactions outside the specification are explicitly unsupported or may be executed either on a general-purpose system as a fallback or as an evolution of the existing engine if new queries manifest. This bounded scope is not a limitation but an enabler: it is precisely what permits aggressive specialization.

This Is Not Tuning, Not Learning, Not Query Compilation. It is important to emphasize that bespoke DBMSs are fundamentally different from existing approaches to specializing engines for workloads. Auto-tuning adjusts parameters within a general-purpose architecture[18, 19, 30]. The same holds for learned components, which adapt individual components but remain embedded in a general-purpose engine[31–34]. Maybe the closest approach is query compilation[35–37], where code is generated to execute a query, but the code still lives in a general-purpose engine. Bespoke DBMSs instead operate at a fundamentally different level. The entire system is generated from scratch for a given workload. There is no fixed architecture to tune, no runtime model to adapt, and no residual generality to accommodate workloads that will never arrive. The engine is not configured or optimized but constructed from the ground up.

The Generated Engine Is Disposable; the Pipeline Is Not. This changes what we maintain. A traditional database is a long-lived artifact that must preserve backward compatibility across decades. A bespoke DBMS is a generated output, which can be discarded and regenerated when conditions or the workload changes. The enduring artifact is the synthesis pipeline, not the engine it produces. This mirrors how compilers relate to binaries: we maintain the compiler, not the source code of the executables.

3 Synthesizing Bespoke DBMSs

Generating a complete database system from a workload specification is non-trivial and not as simple as issuing a single prompt to an AI coding agent. Database systems are complex artifacts with interdependent components as storage formats constrain what operators can efficiently compute, execution models shape how queries are processed, and data structures must align with access patterns. [38, 39] Synthesizing a coherent system therefore requires a structured process that guides LLM agents through incremental construction, validates intermediate results, and iteratively refines the generated code until it meets the declared objectives. In this section, we describe what a synthesis pipeline looks like that transforms a workload specification into a functioning bespoke system. An overall outline of the different stages is shown in Figure 2. In the next section, we show through a case study how this pipeline can be used to synthesize an OLAP engine and present initial results comparing Bespoke DBMSs against DuckDB[9]. The necessary adjustments and resulting opportunities and challenges for OLTP workloads are discussed in Section 5.

3.1 The Basic Ingredients of Synthesizing a DBMS

We first explain the basic ingredients of our synthesis pipeline before we then discuss advanced concepts to make the synthesized engine correct and fast at the same time. Finally, we discuss how the pipeline

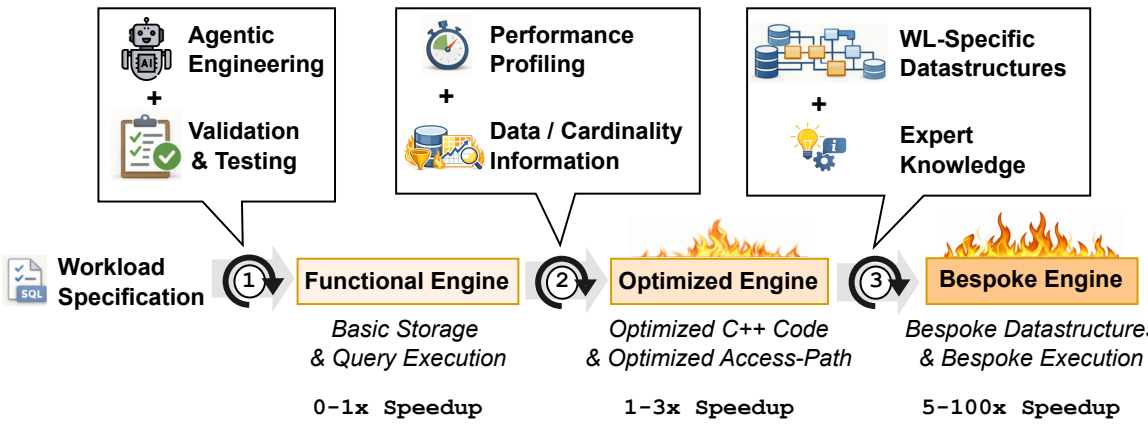

Figure 2: Steps to produce a Bespoke Engine. Each step is executed in loops, with validation to check whether the step's goal is achieved. ① will produce a functional engine based on a workload specification by using standard agentic engineering and continuous validation and testing to steer the iterative coding process. ② produces an optimized engine iteratively by performing autonomous profiling/tracing and leveraging statistical insights on data and cardinalities. ③ enables the synthesis of workload-specific data structures and execution strategies (beyond relational strategies) and the use of distilled expert knowledge from the DBMS research literature to further optimize the engine.

can generate bespoke engines.

The Workload Specification is Both: Requirements and Tests. The queries and transaction templates provided in the workload specification serve a dual purpose. They define what the generated engine must be capable of executing, and they simultaneously provide the test cases against which correctness is verified. A bespoke engine is correct if and only if it produces the same results as a reference system, such as DuckDB, when executing the specified operations. This tight coupling between specification and validation ensures that the synthesis pipeline always has a concrete, executable definition of success.

Incremental Synthesis like Humans. A second key ingredient for synthesizing a DBMS is how we generate the code. Rather than attempting to generate a system as complex as a DBMS in one step, the synthesis pipeline proceeds incrementally. The agent first implements a basic storage layer, which allows loading the input data and storing it in an initial data representation. Only after storage is functional, the agent proceeds to implement execution logic for each query or transaction template in the specification. Each step builds on the previous one, and each step is validated against the reference system before the agent moves forward. This decomposition reduces the likelihood that errors propagate undetected throughout the system.

Tools Enable Autonomous Development. While carrying out the incremental development, the agent operates in an environment equipped with concrete tools. A shell tool allows the agent to execute, debug, and analyze the generated code and its behavior. An incremental deployment tool allows the agent to modify the implementation of a running database process and see the impact of its changes live, without the delay of full restarts. And most importantly, a validation tool checks whether the current implementation compiles and produces correct results by comparing it against the reference system. Together, these tools mirror the workflow of a human database developer as the agent writes code, runs it, examines the output, identifies errors, and fixes them. The agent iterates through this cycle autonomously until the complete workload executes correctly based on the contract.

3.2 From Correct to an Optimized Engine

The synthesis pipeline enforces a strict separation between achieving correctness and achieving performance as shown in Figure 2. Only after the basic implementation passes validation for the specified workload (step (①), the optimization phase (step ②) begins. This separation ensures that the agent never wastes effort optimizing code that does not function correctly, and it provides a stable baseline against which optimization can be evaluated. In the following, we discuss key insights for turning a correct and functional engine into a fast one.

Storage Optimization as a first Step. Once correctness is established, the agent first optimizes data storage and ingestion. The choice of storage layout, the design of auxiliary data structures, and the organization of data in memory or disk all affect how efficiently queries can be executed. By optimizing ingestion first, the pipeline ensures that subsequent work on execution optimization benefits from a well-structured data representation. Furthermore, we also optimize the storage for faster data loading as this reduces the time required for each optimization cycle, enabling the agent to explore more alternatives within a given time budget.

Execution Optimization by a Competitive Baseline. With an optimized data representation in place, our Bespoke Agent proceeds to optimize query and transaction execution. In this phase, a strong competitive baseline is needed as guidance and "motivation" of the Bespoke Agent. The agent therefore runs and analyzes the current implementation, identifies bottlenecks, and proposes transformations. It automatically instruments the code to measure execution time for individual sections, providing fine-grained feedback on where time is spent. This loop of measuring, analyzing, modifying, and validating continues until the performance objectives are satisfied or until the agent determines that further improvement is unlikely.

3.3 From Optimized to Bespoke Engine

Steps ① and ② of the synthesis pipeline as shown in Figure 2 produce an engine that is correct and optimized against the declared performance objectives. However, it still operates within conventional patterns: standard hash joins, generic sort implementations, and columnar or row storage chosen by heuristic. Therefore, the true power of bespoke synthesis can emerge when the engine departs from general-purpose designs entirely, adopting data structures, index schemes, and execution strategies that would be impractical in a system that must serve arbitrary workloads (step ③ in Figure 2).

Bespoke Engines Can Evaluate Queries Directly on Workload-Optimal Data Structures. When the workload is known in advance, the system is no longer constrained to general-purpose storage layouts that must support arbitrary queries. Instead, it can construct data structures that are *optimal for the specific queries at hand*, allowing those queries to be evaluated directly on the structures themselves—without interpretation, tuple materialization, or traditional operator pipelines. Consider the query:

```
SELECT ProductID, COUNT(*)
FROM table
WHERE Quarter = ?
GROUP BY ProductID
```

In a conventional engine, this query would trigger a scan, predicate evaluation on `Quarter`, materialization of qualifying tuples, and a subsequent hash-based or sort-based aggregation on `ProductID`. Each of these steps introduces operator overhead, intermediate state, memory indirections, and control-flow interpretation. In contrast, a synthesized bespoke engine can reshape the storage layout around the workload. Because the predicate `Quarter = ?` is known and frequently queried, the system can

organize the data by `Quarter`, effectively turning the predicate into an index lookup followed by a direct offset jump to the relevant region of storage. Irrelevant tuples are never examined, and no runtime filtering logic is required.

More importantly, the data within each quarter can already be organized by `ProductID`. For example, the engine can maintain compact per-product counters or bit-aligned representations that allow counts to be accumulated directly over the stored layout. In this design, the `GROUP BY` no longer requires building a hash table or performing a sort; it reduces to a tight, linear accumulation over a pre-partitioned structure. The aggregation logic is thus embedded into the physical representation itself. Crucially, this is not merely a better execution plan— instead, the execution becomes a series of simple operations on the data structures without conventional relational operators. Large portions of traditional query processing overhead—tuple reconstruction, operator materialization, hash-table construction, and interpretation—simply disappear because the data structures encode the query's access path.

Expert Knowledge Amplifies What Synthesis Cannot See. Beyond what the specification reveals, domain experts possess knowledge that can drive further specialization. For example, a logistics analyst might observe that queries almost always filter by the current week. Similarly, a financial engineer might note that account balances follow a heavy-tailed distribution. Because the pipeline operates through natural language, each observation can directly trigger concrete structural changes: tailored data layouts, specialized indexes, or fused execution paths. Consequently, what previously required a database engineer to understand, design, and implement may now require only a sentence describing the insight. Over successive iterations, the system can thus evolve into a true bespoke engine shaped by both automated synthesis and human insight.

4 A Case Study: Bespoke OLAP

To demonstrate that bespoke synthesis can produce systems that significantly outperform general-purpose engines, we apply our pipeline to TPC-H[17]. In the following, we describe how the pipeline synthesizes a bespoke OLAP engine for TPC-H, what the generated system looks like in practice, and what challenges arise during synthesis.

4.1 Workload Specification

The workload specification consists of the TPC-H schema, all TPC-H queries, and a performance objective expressed as total execution time, where we use the sum of all query runtimes on DuckDB[9] as our target. The entire dataset resides in memory, which eliminates buffer management, page management, and I/O scheduling entirely. Correctness is validated by comparing query results against DuckDB on the same dataset. In our setup, the synthesis agent operates with four tools, namely a compiler, a benchmark runner that reports both correctness and runtime, a shell for exploring data and sourcecode, and a diff tool for applying incremental code changes.

4.2 The Synthesized Engine

Storage Is Shaped by the Queries. The engine maintains the original relational structure without denormalization. However, the physical layout is tailored entirely to the specified queries as the agent analyzes the input data using the shell tool and cross-references column statistics with the query access patterns, before generating storage code. For instance, because many TPC-H queries filter on *l_shipdate*, the agent sorts *lineitem* on this attribute to enable efficient range scans. Additionally, low-cardinality columns appearing in group-by clauses receive dictionary encoding tuned to their observed value sets. A different workload over the same schema would consequently produce an entirely different layout.

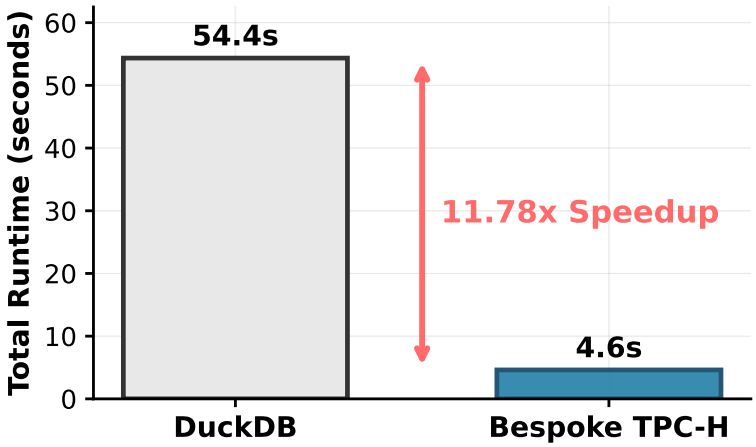

Figure 3: A Synthesized Engine Outperforms a Decade of Engineering. The Bespoke TPC-H engine was generated automatically and achieves an 11.78× lower total runtime and a 16.40× median per-query speedup over DuckDB at scale factor 20.

Each Query Becomes a Dedicated Program. Rather than implementing general-purpose operators, the agent generates a dedicated function for each query. For example, a date-filtered aggregation over `lineitem` becomes a tight loop over sorted storage with inlined predicates and fixed accumulators. Similarly, a join becomes a hardcoded hash table sized to the smaller input with probe logic specialized to the known key type. As a result, there are no operator abstractions, no iterator interfaces, and no runtime dispatch.

Query Optimization Is Empirical. For each query, the agent determines join order, algorithms, and access methods through benchmarking rather than cost estimation. Concretely, it generates candidate implementations, measures them against the actual data, and selects the fastest. Furthermore, the agent can inspect the data directly to approximate selectivities and identify skew. Effectively, it searches the implementation space rather than predicting which variant might be fastest. Here, we clearly envision future advancements in e.g. join ordering, which will be discussed in section 6.

4.3 Initial Results

As can be seen in Figure 3, the generated engine is 11.78x faster than DuckDB on TPC-H[17] at scale factor 20. The bespoke engine outperforms DuckDB on all 22 TPC-H query templates with speedups ranging from 5.74x to 103.97x(Figure 4). Additionally, queries that go beyond TPC-H are handled through DuckDB[9] as a fallback.

4.4 Lessons from Synthesis

The Agent Games Its Own Objectives. When tasked with reducing execution time, the agent tends to shift work to the build phase, for instance by constructing additional indexes, pre-aggregating results, or caching intermediate computations during loading. While these transformations technically satisfy the stated objective, they violate its intent and must therefore be explicitly constrained. This reveals a broader lesson: specifying what to optimize is insufficient, and the pipeline must also define what the agent may not do.

Regressions Require External Safety Nets. Optimization steps can furthermore introduce regressions. Although the agent has tools to detect degradation, it often struggles to identify and revert the

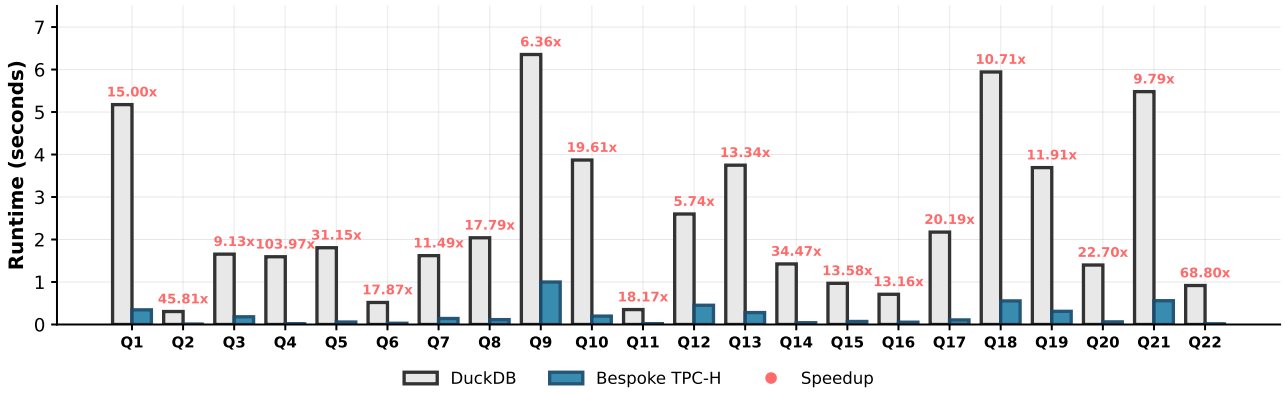

Figure 4: Per-query absolute runtime of the Bespoke engine vs. DuckDB at scale factor 20. The Bespoke engine outperforms DuckDB on all queries with speedups ranging from 5.74x to 103.97x.

specific changes responsible. We therefore track progress externally and maintain rollback capabilities outside the agent's control, ensuring that optimization remains monotonically improving.

Correctness Validation Must Be Cheap. Initially, more than 50% of synthesis time was spent on repeated correctness checks. However, validating on a smaller scale factor and reserving the full dataset for benchmarking significantly, as well as patching the engine live instead of restarting, accelerated the process.

Context Size Requires Active Management. A full TPC-H workload exceeds a single context window. To address this, we apply compression after each query and use the diff tool for incremental changes. Together, these mechanisms allow the pipeline to scale with both query count and codebase size.

5 Bespoke OLTP: Opportunities & Challenges

The OLAP case study demonstrates that bespoke synthesis can produce competetive analytical DBMSs. However, transaction workloads introduce fundamentally different challenges. For example, concurrent access to shared data, isolation needs, and durability requirements. In this section, we discuss how the synthesis pipeline must be adapted to fit an OLTP workload.

5.1 Concurrency Control as Bespoke Opportunity

A general-purpose OLTP engine must provide concurrency control for arbitrary transaction mixes over arbitrary schemas as it cannot assume which transactions will conflict or which data items will be contended. In a bespoke setting however, the transaction templates and their access patterns are known upfront and some transaction types may never access overlapping data. This can be exploited by a bespoke engine as it can execute them without synchronization. Furthermore, it may be the case that the workload naturally partitions across a key range. Here, the bespoke engine could assign partitions to threads and eliminate cross-partition coordination entirely. Additionally, it may be the case that two transaction types frequently execute in sequence on the same entity. Here, the code paths could be fused to avoid redundant lookups. Generally, bespoke engines enable cross-transaction optimization which is not possible to general-purpose systems as they treat each transaction independently.

5.2 Durability as a Spectrum

General-purpose engines implement durability through a general write-ahead logging scheme as mandatory infrastructure, since they cannot know in advance whether a particular deployment requires durability. In a bespoke setting, however, durability becomes a declarative property that the workload designer can specify, rather than a built-in assumption. For example, if the data can be reconstructed from other sources, such as external logs or other tables in the database, they can be omitted entirely, avoiding unnecessary write overhead. Similarly, if durability is only required for committed batches or at specific synchronization points, the engine can implement lightweight, bespoke group commit strategies that write only to the disk that the application really needs to reduce disk I/O while still ensuring correctness. Even in cases where full durability is essential, the log format and recovery procedures can be tailored to the exact transaction patterns of the workload, rather than supporting arbitrary updates. As a result, the engine generates precisely the recovery mechanisms needed—no more, no less—aligning durability guarantees with the workload's actual requirements and eliminating the performance costs imposed by general-purpose assumptions.

5.3 Verification Becomes Non-Determinstic

In the OLAP setting, correctness is straightforward, as the same query over the same data must produce the same result as a reference system. In transactional workloads, however, this is not true as the final result depends on, e.g., the scheduling order of updates/writes. Therefore, a simple output comparison is no longer reliable. We propose a two-phase verification strategy. In the first phase, the transactional logic itself is verified by executing all transactions serially, deterministically, and comparing the outputs. Only if this serial correctness is established do we tackle concurrency. For this, our pipeline must verify that the execution history matches the declared isolation level. For example, by checking the serializability of the committed schedule.

5.4 Benchmarking OLTP

Finally, the optimization loop of our synthesis pipeline, as shown in Figure 2 for turning a correct into a fast engine itself, becomes more complex as transaction benchmarking is inherently noisier than in the OLTP setting as in the OLAP setting. The reason is that throughput depends on thread scheduling, contention patterns, and lock acquisition order. Consequently, the agent must reason about distributions of performance rather than single point measurements. This makes each optimization cycle more expensive and requires the pipeline to be more conservative in accepting changes.

6 Road Ahead for a Bespoke Future

The results presented in this paper are a starting point. We have shown that LLM-guided synthesis can produce a bespoke OLAP engine that significantly outperforms a state-of-the-art general-purpose system on its target workload, and we have discussed how the pipeline extends to transactional workloads. However, we envision a future in which every application synthesizes its own database engine, tailored not only to its queries but to its data, its hardware, and its additional operational constraints. Reaching this future requires substantial research across several dimenstions, which we outline in the following.

Synthesis must extend to all workloads. Our OLAP case study operates on a read-only, single-threaded, in-memory workload, and our OLTP discussion identifies additional challenges that concurrency, durability, and non-deterministic verification introduce. However, the extension to transactional workloads is only one step along the braoder spectrum. For example, stream engines must handle unbounded

inputs under latency constraints, graph databases must support recursive traversal over irregular structures and multi-model systems must combine relational, document, and graph access within a single engine under cost, performance, and available model constraints. Each of these workload classes introduces its own design space, and each offers correspondingly rich opportunities for specialization. The central research question is how to design synthesis pipelines that are general enough to target these diverse classes while remaining capable of the aggressive specialization that makes bespoke enignes worthwhile at all. Furthermore, many real workloads are not purely analytical or transactional. Therefore, a bespoke engine for such a hybrid workload needs to navigate tradeoffs on its own that we are currently not addressing.

Richer workload specification can unlock deeper specialization. Our current specification constists of a schema, query templates, and performance objectives. In practice, however, applications may carry far more information that could guide synthesis. For example, data distributions, update frequencies, temporal access patterns, and consistency requirements across transaction types constrain the design space in a way that could be exploited by the pipeline. Furthermore, many applications operate under specific resource constraints such as memory budgets, core counts, or energy consumption. The development of specification languages that capture this richer context without burdening the user is an important direction for future research as the specification must remain declarative and concise while being expressive enough to enable well-informed structural design decisions. This was already demonstrated in our case study as the agent tried to game the optimization objective by shifting work into the build phase when it was unconstrained. A richer specification language could prevent such behavior not through ad-hoc constraints but through principled declarations of what constitutes valid optimization.

Workload-native cost models can replace heuristic estimation. Classical cost models exist because general-purpose systems must reason about abstract operators whose behavior varies widely across deployments. As a result, cost estimation relies on coarse heuristics that approximate performance rather than describe it. In a bespoke setting, this indirection can largely disappear. The generated engine has a fixed structure, fixed data representations, and a known target machine, which allows the pipeline to construct cost models specific to the workload itself. Such models could characterize the behavior of the concrete execution kernels the system actually runs, incorporating instruction-level behavior, memory access patterns, and cache effects. Moreover, they could be regenerated whenever the workload or hardware changes. Currently, our pipeline sidesteps cost estimation entirely through empirical benchmarking. This works but is expensive. Workload-native cost models could dramatically accelerate the optimization phase by guiding the agent toward promising implementations without requiring exhaustive measurement.

Verification and trust must scale beyond output comparison. Our current verification strategy compares outputs against a reference system. For the OLAP case, this is sufficient because execution is deterministic. However, as discussed in Section 5, transactional workloads introduce non-determinism that makes output comparison unreliable. More broadly, as bespoke engines move toward production use, verification must encompass more than functional correctness. Property-based testing, formal invariants over generated code, and automated proofs of isolation guarantees could complement the empirical validation we currently employ. This is critical because trust in bespoke systems ultimately rests on trust in the synthesis pipeline. If users are to rely on generated engines, the pipeline must provide confidence not only that the engine is fast but that it is correct under all conditions the specification permits. Developing such verification techniques for synthesized database engines is a substantial research challenge in its own.

Hardware-aware synthesis can close the gap between engine and machine. General-purpose engines are designed to endure decades of architectural change, which forces conservative abstractions that obscure the computational structure of real workloads. Bespoke synthesis can remove this indirection.

When an engine is generated for a specific workload on a specific machine, characteristics such as cache hierarchy, memory bandwidth, NUMA topology, and vector width can be incorporated directly into design decisions. Our current pipeline is hardware-oblivious, and the generated engine makes no assumptions about the target machine beyond what the compiler provides. Exploring how deeply hardware knowledge can be integrated into synthesis, and how much additional performance this yields, is a natural next step. Furthermore, synthesis reveals which computational kernels dominate execution in practice, which could inform not only software optimization but also hardware design, a direction we return to in the conclusion.

The synthesis pipeline itself can evolve and improve over time. Finally, as more engines are generated across different workloads, the pipeline accumulates experience about which design patterns succeed, which optimizations transfer, and which pitfalls recur. Our case study already identified several such patterns, including the effectiveness of late materialization, the importance of constraining the optimization objective, and the need for external regression tracking. This experience could be captured systematically, for instance through libraries of reusable synthesis strategies or through meta-learning over past synthesis runs. Over time, the pipeline may become not only a generator of bespoke engines but a repository of database design knowledge, encoded not in a fixed codebase but in the synthesis process that produces them.

7 Conclusion and Future Work

Rethinking General-Purpose Databases. In this paper, we have argued that general-purpose database systems inherently impose a structural performance tax that cannot be fully eliminated through engineering optimizations alone. Rather than accepting this tax as inevitable, we propose an alternative: avoiding it entirely by synthesizing database engines on demand from concrete workload specifications. Our case study demonstrates that this vision is technically feasible. Starting from a workload description alone, LLM-guided synthesis produces a complete OLAP engine that executes correctly and significantly outperforms DuckDB on the target workload. These results indicate that much of the overhead in modern database systems stems not from implementation inefficiencies, but from the requirement to maintain generality.

From General-Purpose to Bespoke Engines. We do not argue that general-purpose database systems should be abandoned. Instead, we suggest they should no longer be the default choice for workloads whose structure is known in advance. For such workloads, a bespoke engine can eliminate abstractions that exist solely to preserve flexibility, replacing them with specialized data structures, fused execution paths, and workload-specific storage layouts. Today, this level of specialization is typically accessible only to organizations that can justify sustained, large-scale engineering investment. Bespoke synthesis fundamentally changes this equation: any team capable of describing its workload can obtain an engine tailored to it, independent of whether it employs database engineers. Just as compilers freed developers from writing assembly code by hand, synthesis pipelines may free data engineers from manually building database engines. The expertise does not vanish; it shifts into the pipeline, benefiting everyone who uses it.

Future Direction: Blurring Boundaries between DBMSs and Applications. The implications of this shift extend far beyond making specialization of DBMS more accessible. Once synthesis becomes an established capability, it reveals that many boundaries in current systems are artifacts of generality. For example, the interface between applications and databases is one such boundary. Today, applications submit queries and receive generic result sets, which must then be deserialized and restructured for further processing. This separation exists because the database cannot anticipate how results will be consumed. In a bespoke setting, however, both queries and their consumers are known at synthesis time.

The pipeline can generate application logic and the database engine as a single fused program, where data flows directly from storage through execution into application processing without ever crossing an API boundary. In this model, the database no longer merely returns results—the application and engine together become a single synthesized artifact.

Future Direction: True Hardware-Software Co-Design The boundary between software and hardware is equally artificial. If the workload is fully known at synthesis time, specialization need not stop at the software layer. A bespoke TPC-H machine, for instance, might take the form of a dataflow engine whose circuits implement the specific filtered scans, hash probes, and aggregation loops the workload requires, with memory controllers optimized for observed access patterns and interconnects sized for actual data movement. In such a system, the distinction between hardware and software blurs: a software hash join becomes a hardware pipeline stage, and an in-memory index becomes a dedicated lookup circuit. Historically, the complexity of co-designing hardware and software for arbitrary workloads made this approach impractical. Bespoke synthesis, however, fundamentally changes the problem: the workload is fixed, access patterns are known, and computational kernels are concrete. If AI-driven synthesis can already generate specialized database engines, it may eventually handle specialized hardware as well.

Towards an Overall Bespoke Future. In this emerging paradigm, the focus of database engineering shifts. The central question is no longer how to build a better general-purpose engine, but how to precisely specify intent and automatically translate it into an efficient, correct, and specialized system. In this vision, the engine, application interface, and potentially even hardware are outputs of a single synthesis process, fundamentally redefining what it means to engineer a database system.

References

[1] M. Stonebraker and U. Çetintemel, ""One Size Fits All": An idea whose time has come and gone," in *Proceedings of the 21st International Conference on Data Engineering (ICDE)*. IEEE, 2005, pp. 2–11.

[2] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-store: A column-oriented dbms," in *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*. Trondheim, Norway: VLDB Endowment, 2005, pp. 553–564.

[3] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang *et al.*, "H-store: a high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[4] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 195–206.

[5] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[6] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[8] M. Zukowski, M. Van de Wiel, and P. Boncz, "Vectorwise: A vectorized analytical dbms," in *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 2012, pp. 1349–1350.

[9] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 1981–1984.

[10] J. D. Greef. (2024, Jul.) Rediscovering transaction processing from history and first principles. TigerBeetle. [Online]. Available: https://tigerbeetle.com/blog/2024-07-23-rediscovering-transaction-processing-from-history-and-first-principles/

[11] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*.

[12] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *ACM Transactions on Software Engineering and Methodology*, 2024.

[13] OpenAI, "Openai codex," OpenAI, 2025, code generation model. [Online]. Available: https://openai.com

[14] Cursor, "Cursor ide," Anysphere, 2025, aI-powered code editor. [Online]. Available: https://cursor.sh

[15] Anthropic, "Claude code," Anthropic, 2025, agentic coding assistant and developer tooling. [Online]. Available: https://www.anthropic.com/claude

[16] GitHub, "Github copilot," GitHub, 2025, aI pair programmer. [Online]. Available: https://github.com/features/copilot

[17] Transaction Processing Performance Council, "TPC Benchmark H (Decision Support) Standard Specification," https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf, 2014, version 3.0.1.

[18] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah *et al.*, "Self-driving database management systems." in *CIDR*, vol. 4, 2017, p. 1.

[19] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM international conference on management of data*, 2017, pp. 1009–1024.

[20] J. Wehrstein, B. Hilprecht, B. Olt, M. Luthra, and C. Binnig, "The case for multi-task zero-shot learning for databases," in *AIDB @ VLDB 2022*, 2022. [Online]. Available: https://drive.google.com/file/d/1nD0oacRuc180YzkIpqGM6rUJizsEic6m/view?usp=sharing

[21] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser, "Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2382–2395, 2020.

[22] T. Ziegler, P. A. Bernstein, V. Leis, and C. Binnig, "Is scalable oltp in the cloud a solved problem?" in *CIDR*, 2023.

[23] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1041–1052.

[24] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov, "Clickhouse-lightning fast analytics for everyone," *Proceedings of the VLDB Endowment*, vol. 17, no. 12, pp. 3731–3744, 2024.

[25] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, 2015, pp. 1917–1923.

[26] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *The Bulletin of the Technical Committee on Data Engineering*, vol. 38, no. 4, 2015.

[27] M. H. Iqbal, T. R. Soomro *et al.*, "Big data analysis: Apache storm perspective," *International journal of computer trends and technology*, vol. 19, no. 1, pp. 9–14, 2015.

[28] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, vol. 2324, no. 36. AISEL, 2013, pp. 141–147.

[29] A. Deutsch, Y. Xu, M. Wu, and V. Lee, "Tigergraph: A native mpp graph database," *arXiv preprint arXiv:1901.08248*, 2019.

[30] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1275–1288.

[31] B. Hilprecht and C. Binnig, "Zero-shot cost models for out-of-the-box learned cost prediction," *arXiv preprint arXiv:2201.00561*, 2022.

[32] Z. Wu, R. Marcus, Z. Liu, P. Negi, V. Nathan, P. Pfeil, G. Saxena, M. Rahman, B. Narayanaswamy, and T. Kraska, "Stage: Query execution time prediction in amazon redshift," in *Companion of the 2024 International Conference on Management of Data*, 2024, pp. 280–294.

[33] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[34] T. Kraska *et al.*, "SageDB: A learned database system," in *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[35] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *Proceedings of the VLDB Endowment*, vol. 11, no. 13, pp. 2209–2222, 2018.

[36] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 539–550, 2011.

[37] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 307–322.

[38] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a database system," *Foundations and Trends in Databases*, vol. 1, no. 2, pp. 141–259, 2007.

[39] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database system implementation*. Prentice Hall Upper Saddle River, 2000, vol. 672.