Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD – Extended Version

Geoffrey X. Yu
MIT CSAIL
Cambridge, MA
geoffxy@mit.edu

Ziniu Wu
MIT CSAIL
Cambridge, MA
ziniuw@mit.edu

Ferdi Kossmann
MIT CSAIL
Cambridge, MA
kossmann@mit.edu

Tianyu Li
MIT CSAIL
Cambridge, MA
litianyu@mit.edu

Markos Markakis
MIT CSAIL
Cambridge, MA
markakis@mit.edu

Amadou Ngom
MIT CSAIL
Cambridge, MA
ngom@mit.edu

Samuel Madden
MIT CSAIL
Cambridge, MA
madden@csail.mit.edu

Tim Kraska
MIT CSAIL, AWS
Cambridge, MA
kraska@mit.edu

ABSTRACT

Modern organizations manage their data with a wide variety of specialized cloud database engines (e.g., Aurora, BigQuery, etc.). However, designing and managing such infrastructures is hard. Developers must consider many possible designs with non-obvious performance consequences; moreover, current software abstractions tightly couple applications to specific systems (e.g., with engine-specific clients), making it difficult to change after initial deployment. A better solution would *virtualize* cloud data management, allowing developers to declaratively specify their workload requirements and rely on automated solutions to design and manage the physical realization. In this paper, we present a technique called *blueprint planning* that achieves this vision. The key idea is to project data infrastructure design decisions into a unified design space (blueprints). We then systematically search over candidate blueprints using cost-based optimization, leveraging learned models to predict the utility of a blueprint on the workload. We use this technique to build BRAD, the first cloud data virtualization system. BRAD users issue queries to a single SQL interface that can be backed by multiple cloud database services. BRAD automatically selects the most suitable engine for each query, provisions and manages resources to minimize costs, and evolves the infrastructure to adapt to workload shifts. Our evaluation shows that BRAD meet user-defined performance targets and improve cost-savings by 1.6–13× compared to serverless auto-scaling or HTAP systems.

1 INTRODUCTION

Over the past decade, the cloud has transformed how organizations manage their data through two key forces: (i) offering a plethora of specialized database engines optimized for diverse workloads [10, 12, 15], and (ii) enabling "one-click" on-demand access to conceptually "infinite" resources [14, 17, 45, 74]. To reap these benefits, cloud users must curate a collection of such specialized database engines, each offering a competitive edge on different parts of their workload. For example, an organization might use Aurora [12] to manage client accounts with transactions, Snowflake [34] to analyze historical sales data, and BigQuery Omni [43] for exploratory analysis. Benefits aside, these multi-system infrastructures introduce new management challenges. Data engineers need to (i) choose a suitable set of engines (out of dozens [20, 21, 44]) for their workload, (ii) partition and/or replicate their data across the engines, (iii) decide which engines to use for each aspect of their workload (i.e., which queries go to each engine), (iv) provision the engines appropriately, and (v) repeat these steps each time their workload or business needs change. Navigating these decisions is hard; prior work showed that an optimal infrastructure depends on many interconnected factors such as query selectivity, service level objectives (SLOs), and dynamic load of the system [60]. Designs based on conventional wisdom can miss out on significant performance and cost savings (Section 2.1). As a result, organizations struggle to design their infrastructure while also keeping costs under control [46].

To address this challenge, we recently presented our vision for BRAD [60]. BRAD is fundamentally a virtualization layer for cloud data infrastructure. BRAD users do not specify the mapping of data to specific engines or explicitly provision resources. Instead, BRAD uses a proxy-like indirection layer [19, 24, 29] to abstract away multiple database engines, appearing to end-users as a single SQL endpoint. Under the covers, BRAD allocates data and operates the infrastructure by picking the "best" set of engines for the workload, choosing the appropriate data distribution and provisioning for each engine, and routing queries optimally. This is a fundamentally challenging because BRAD must explore a huge space of possible solutions, while meeting performance expectations.

We solve this problem using a novel technique we call *blueprint planning*, which is a holistic *cost-based optimization* over the infrastructure design space. Specifically, blueprints are system plans that define a BRAD deployment. They contain (i) the set of engines to include in the infrastructure, (ii) their provisioning configurations (e.g., instance type and number of nodes), (iii) the engine(s) on which each table in the dataset is placed, and (iv) a policy for routing queries to the engines. Blueprints allow us to systematically and quantitatively consider all aspects of the infrastructure design problem in a unified search space, analogous to traditional query planning [96]. However, accurately assigning scores to blueprints is significantly harder than query planning. First, the utility of a blueprint is not captured by performance alone, as a good blueprint for a given workload minimizes dollar-based operating costs under a latency-based performance constraint (or vice-versa, depending on user-specified goals). Second, accurately predicting a workload's performance (e.g., a query's run time) on a blueprint is difficult due to (i) engines having opaque system implementations and (ii) new constraints in our setting. Specifically, we must make these predictions when a physical query plan is unavailable, preventing us from reusing existing learned models [50, 70, 71, 73, 100, 114]. For

example, a candidate blueprint may add an engine into the infrastructure that is not yet running (e.g., starting up a data warehouse) or replicate a table onto a new engine to support a query.

In this paper, we show that these challenges are tractable. In the cloud setting, infrastructure operators can collect performance data over a wealth of workloads and deployments to build learned performance models. Moreover, most query optimizers are deterministic. Thus, we can train a model to predict a query's run time using just its logical properties (e.g., filter selectivities, join templates) since the optimizer will pick similar query plans with comparable run times for similar queries. We leverage these observations to build a graph neural network with a novel query featurization that relies only on such logical query features (Section 3.2). Together with other analytical models, we use this model to predict the performance and cost of candidate blueprints on a given workload. We then use these predictions to drive a greedy beam-based search over the blueprint search space to find an optimized infrastructure design. We have implemented our blueprint planner in BRAD, enabling it to automatically design infrastructures consisting of three engines that cover a large part of enterprise needs: (i) a transactional store (Aurora [12]), (ii) a data warehouse (Redshift [15]), and (iii) a data lake query engine (Athena [10]).

While there is a wealth of prior work in automatically optimizing single systems [68, 78, 82–84, 86–88, 95, 106], and in managing existing multi-engine deployments [5, 25, 27, 28, 37, 41, 52, 53, 92, 97, 110, 117], BRAD holistically automates and optimizes the design and operation of multi-engine infrastructures. Doing so involves reasoning about cost and performance across engines and hypothetical deployments, which, to our knowledge, have not been studied.

We evaluate BRAD by having it automatically optimize a data infrastructure for cost under a performance constraint, We use a workload with both transactions and diverse analytics running on an adapted version of the IMDB dataset [63]. Overall, we show that BRAD is able to react to changing workloads and select designs that achieve performance targets in diverse deployment scenarios. When compared to a baseline that naïvely auto-scales transactional and analytical systems, BRAD achieves 1.6–13× cost savings due to its ability to route queries between engines and precisely scale to the resource needs of a workload, instead of reacting passively to increased system load.

Contributions. In summary, we make the following contributions:

- We introduce *blueprint planning*: a new framework for virtualized, automated cloud data infrastructure design and management that applies cost-based optimization.
- We present a practical blueprint planning solution. We leverage a graph neural network with a novel logical query featurization that generalizes to common gradual workload changes.
- We present the design, implementation, and evaluation of BRAD: a virtualized cloud database management system that uses blueprint planning to automate infrastructure design.

2 CONQUERING THE COMPLEX CLOUD

We first illustrate the subtle challenges in cloud infrastructure design and contrast this experience with using BRAD.

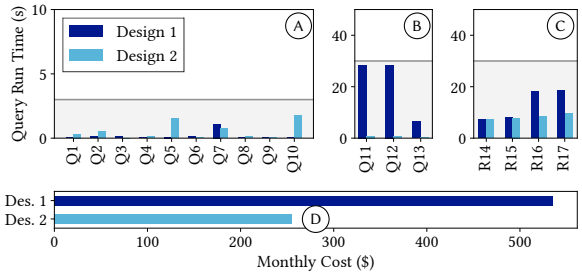

Figure 1: Query performance and operating costs of the same workload on two data infrastructure designs.

2.1 When Conventional Wisdom Falls Short

Consider the data processing needs of a movie theater chain. Under conventional wisdom, they should run an OLTP engine (e.g., Aurora) for their transactions and a data warehouse (e.g., Redshift) for their analytics. To show the downsides of this approach, we run a synthetic workload based on a 160 GB version of the IMDB dataset [63] comprising transactions, repeating dashboarding queries Ⓐ Ⓑ, and periodic reporting queries Ⓒ. We run Aurora with one db.t4g.medium instance and two Redshift dc2.large nodes (Design 1). Figure 1 shows the analytical query latencies. We aim to keep some queries under 3 s Ⓐ and others under 30 s Ⓑ Ⓒ.

At first glance, Design 1 appears reasonable. However, consider an alternative design with just two Aurora db.t4g.medium instances: a primary and replica (Design 2). We can run a subset of the queries Ⓐ Ⓑ on the Aurora replica and offload the reporting queries Ⓒ onto Athena (a serverless data lake engine). As shown in Figure 1, Design 2 saves 2× on cost Ⓓ, meets the performance targets, and even improves query latency on some queries (up to 48× Ⓑ and 2.1× Ⓒ). Transaction latency is unaffected on both designs because they run on the unchanged Aurora db.t4g.medium primary instance.

Design 2 performs better because some queries Ⓑ have predicates on indexed columns, which Aurora can leverage. Redshift does not support indexes and must use table scans. The reporting queries Ⓒ run infrequently, once every four hours, so they can be offloaded to Athena (a serverless engine) instead of incurring a high cost on a provisioned but underutilized Redshift cluster. Athena's serverless burst capability enables the up to 2× decrease in query latency Ⓒ. These queries cannot meet the performance targets on Aurora; they would run for over 150 seconds each.

This example shows that an effective design strongly depends on the specifics of the workload and engines, rather than high-level guiding principles (e.g., run transactions on an OLTP engine and analytics on a data warehouse). Here, the conventional wisdom design is about twice as expensive and an order of magnitude slower on some queries than Design 2. An engineer would need an intimate understanding of the engines and workloads to find such a design, possibly spending a lot of time doing so. Moreover, because the best design is workload-dependent, it can change in response to workload shifts, forcing the engineer to redo their work. These repeated design endeavors are unscalable and difficult to get right, underscoring the need for a principled and automated alternative.

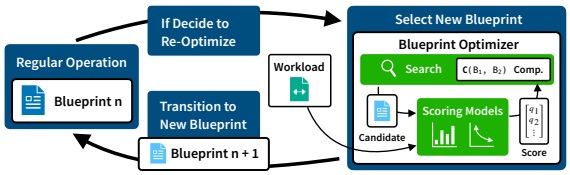

Figure 2: BRAD's blueprint planning life cycle.

2.2 BRAD to the Rescue

With BRAD, users no longer manually design and operate their infrastructures. Instead, each virtualized database engine managed by BRAD has a user-specified design goal (e.g., minimize cost and keep query latency under 30 seconds), and users simply submit their queries and transactions directly to BRAD as if it were a single engine. BRAD uses this design goal to optimize the infrastructure to best run the user's workload—what we call *blueprint planning*. In this paper, we focus on the models, algorithms, and mechanisms used in BRAD's blueprint planner (Section 3). That said, virtualization comes with many more challenges. In the remainder of this section, we briefly outline how BRAD tackles them. We leave a more detailed exploration of this topic for future work.

Data consistency and freshness. Since BRAD can choose to back a (virtual) table with multiple replicas across engines, consistency and freshness are natural concerns. In BRAD, (i) a table T will have exactly one "writer engine" E (i.e., all DML statements affecting T run on the same E), and (ii) transactions run on a single engine (Aurora). BRAD syncs its table replicas (if any) at a user-defined frequency. Analytical queries (i.e., read-only queries not part of a transaction) run against a snapshot, but the snapshot can be stale up to the last sync. All transactions always run on the latest snapshot. This approach provides similar freshness to existing solutions [55].

Transformations. Modern data infrastructure designs typically use ELTs [7, 99], meaning that tables are first replicated into a data warehouse and then transformed inside the warehouse using DML statements. BRAD supports this model, as it already syncs table replicas across engines; these transformations would thus run as regular DML statements that modify the logical tables in BRAD. Running these transformations on a schedule is orthogonal to BRAD and can be handled using an external tool.

SQL dialects and semantics. Different database engines can have different SQL dialects and semantics, meaning the same SQL statement may not be executable on every engine. Currently, we assume that BRAD can detect the subset of its engines that can correctly run a given SQL query; BRAD will ensure it only routes the query to those eligible engines (see Section 4.3). SQL dialect translation techniques [23] may enable BRAD to expand a query's set of eligible engines; we leave this to future work.

3 BRAD'S BLUEPRINT PLANNER: KEY IDEAS

We now describe the key ideas behind BRAD's blueprint planner. We continue with further implementation details in Section 4.

3.1 The Blueprint Planning Life Cycle

BRAD automatically designs and operates data infrastructures using a blueprint planning *life cycle*, which we depict in Figure 2. The core idea is to select the "best" blueprint for the user's workload, operate the infrastructure according to the blueprint, and then trigger re-optimization if the workload changes (Section 4.6). Concretely in BRAD, blueprints are infrastructure plans that contain

- The engines to include in the underlying data infrastructure.
- The provisioning configuration to use for each engine when applicable (e.g., instance type, the number of nodes to use).
- The placement of data tables and replicas on the engines.
- A policy for routing queries to the engines in the infrastructure.

Below, B is an example blueprint describing an infrastructure comprising Aurora (provisioned with one db.r6g.xlarge instance) and Athena. Table T_1 is placed on Aurora, and T_2 is replicated on Aurora and Athena. The routing policy consists of concrete query assignments chosen during blueprint optimization (query q_1 to Aurora and q_2 to Athena) (see Section 3.3) and an online policy $P(q)$ that selects an engine for a given query q (see Section 3.4).

$$B = \begin{cases} \{\text{Aurora, Athena}\} & \textbf{Engines} \\ \{(\text{Aurora, db.r6g.xlarge}, 1)\} & \textbf{Provisioning} \\ \{T_1 \rightarrow \text{Aurora}, T_2 \rightarrow \text{Aurora}, T_2 \rightarrow \text{Athena}\} & \textbf{Placement} \\ \{q_1 \rightarrow \text{Aurora}, q_2 \rightarrow \text{Athena}, P(q)\} & \textbf{Routing} \end{cases}$$

To automatically find an optimized blueprint, BRAD needs a mechanism to (i) quantify the utility of (i.e., assign a "score" to) candidate blueprints on the user's workload (Section 3.2), and (ii) to systematically search over the blueprint design space (Section 3.3). Here, a workload is a representative (but not necessarily exhaustive) list of expected queries and DML statements along with dataset statistics (e.g., its size). Concretely, BRAD obtains a user's workload by logging their transactions and queries (see Section 4.1).

Scoring a blueprint is challenging because multiple factors influence a blueprint's utility (e.g., performance, cost), and different users may have different design goals (e.g., maximizing performance vs. minimizing cost). Consequently, BRAD assigns *vector scores* to its candidates, which comprise three components:

(1) **Workload Performance.** BRAD predicts the run time of the queries and transactions in the workload on the blueprint.
(2) **Operating Cost.** The monetary cost of operating the data infrastructure and routing policy specified by the blueprint.
(3) **Transitions.** The time and monetary cost of *transitioning* the underlying infrastructure to the candidate blueprint.

For example, a vector score would look like $[q_1\ q_2\ \ldots\ c\ t_T\ c_T]^\intercal$ where the q_is are predicted query and transaction latencies, c is the operating cost, and t_T and c_T are the transition time and cost. BRAD uses a set of learned models to assign values to all three components, which we discuss next in Section 3.2.

Users express their "design goals" to BRAD by providing a comparator function that ranks the vector scores (Section 4.5), analogous to the comparators used in sort routines [33]. For example, one such goal could be to design an infrastructure that minimizes cost while maintaining a performance constraint (e.g., a latency SLO, see Section 4.5). The comparator would therefore rank blueprints by their operating cost while treating blueprints that are predicted to not meet the latency constraint as having an infinite cost.

3.2 Blueprint Scoring

In the blueprint's score vector, the main challenge is predicting the performance of the workload on the blueprint. For BRAD, this means estimating the latencies of the queries in the workload on each of BRAD's engines while taking into account their provisioning and load. We discuss scoring in more depth in Section 4.2.

3.2.1 Query Run Times. Prior work has proposed run time prediction methods for use in query optimization [100], workload scheduling [109], resource management [95], and maintaining SLOs [32]. These methods require the query's physical execution plan as input. For example, DBMS cost models and traditional predictors use hand-derived heuristics to understand the cost of each physical operator [4, 38, 65, 111]. Advanced methods featurize the physical query plans and train deep learning models to predict their run time [50, 70, 71, 73, 100, 114]. BRAD cannot directly use these methods because it cannot always get a physical plan. For example, BRAD may need to predict the run time of a query on an engine that is not running or does not have the relevant data loaded (e.g., to decide whether to start Redshift and/or move a table there). BRAD must also account for the effects of provisioning and system load.

BRAD addresses these challenges using a graph neural network (GNN) and two analytical models. We design a new GNN that predicts a query's run time using the query's SQL as input (i.e., relies only on logical features) for an unloaded engine on a fixed provisioning. Our GNN's novelty is that it featurizes a query based on its SQL text and data properties. In contrast, existing models featurize queries using their physical query plans. We then use two analytical models, based on Amdahl's law [22] and queuing theory [48] to adjust this model's estimates for different provisionings and system loads, respectively. We take this approach because making such predictions with a single model is expensive and hard to realize due to the need for diverse run time observations across various query types, provisionings, and system loads. We describe the details of our analytical models in Section 4.2.2; they provide an acceptable accuracy and enable BRAD to find effective blueprints (Section 5.2).

GNN model and query featurization. We use a GNN model with a novel query featurization that depends only on logical query properties (e.g., the join template, join/filter selectivity) and dataset statistics (e.g., estimated join selectivity). Our design is based on the key observation that most query optimizers are deterministic: they will choose similar query plans with similar run times for queries with similar features and statistics. Thus, we identify these features and then model them with a novel graph structure. As a result, even without physical plans, our model learns the optimizer's behavior and makes accurate predictions for queries similar to the training queries in our featurization space (Section 5.3). We use the same approach to predict the amount of data a query scans (to estimate Athena's query cost). We describe the featurization and graph structure in more detail in Section 4.2.1.

3.2.2 Model Bootstrapping. BRAD is designed to be gradually deployed onto an existing infrastructure running one or more of our component engines. When first deployed, BRAD observes the running workload and gathers performance data (e.g., query run times) for each engine in a brief "bootstrapping phase." BRAD then uses this data to train these aforementioned models. Once complete, BRAD then begins to actively optimize the infrastructure using its blueprint planner. Avoiding a bootstrapping phase would require having performance models that are fully transferable across workloads and datasets, which we leave to future work.

3.3 Blueprint Search

Exhaustively searching the entire design space is intractable for most workloads because it is exponentially large with respect to the number of tables and queries. Given this challenge, BRAD must use an efficient search algorithm that finds optimized blueprints without visiting the entire search space.

BRAD uses a greedy beam-based search [69] over the blueprint's routing policy (i.e., query-to-engine mapping), which directly impacts the workload's performance. Blueprints are optimized for a workload which contains a representative list of expected queries. The idea is to incrementally expand a set of top-k blueprints (the "beam") by examining queries in the workload one-by-one. For each blueprint in the current top-k, the planner takes the next query and assigns it to each of the three engines, creating three new candidate blueprints. It then places tables according to these routing decisions (e.g., if a query accessing table T is routed to E, then a copy of T is placed on E). After each step, the planner only keeps the top-k blueprints, and it repeats until all the queries have been assigned. BRAD runs this beam search for each provisioning "near" the current one (in computational resources) and returns the best-scoring blueprint. BRAD uses a beam of size 100 (i.e., $k = 100$). We analyze and discuss our search algorithm in more detail in Section 4.4.

3.4 Operating Blueprints: Query Routing

Once BRAD has chosen a blueprint, the final step is to use it to serve the workload. To route queries, BRAD first consults the query-to-engine assignment in its blueprint. If the query is in the assignment, BRAD uses this pre-planned routing decision. Otherwise, it uses an online routing policy. The key challenge in designing this policy is that it must make intelligent routing decisions without imposing an undue overhead (i.e., doing so within tens of milliseconds). This efficiency constraint precludes the use of computationally expensive models, such as the query run time model we use for blueprint scoring. To address this challenge, we make two key observations. First, in tasks like run time prediction, prior work showed that classification is easier than regression in terms of model efficiency and accuracy [39, 119]. Thus, we can cast this routing problem as a classification problem and leverage a lightweight classifier. Second, this online routing policy can be trained during blueprint planning, which runs off of the critical path. This workflow gives us the opportunity to bootstrap the online routing policy using a more sophisticated but computationally expensive model, e.g., our query run time model.

We leverage these observations to design BRAD's online routing policy. BRAD's online policy is a decision forest that, for a given query, produces a ranking of the engines (most preferred routing to least). BRAD routes the query to the highest-ranked engine that has all the tables the query accesses. As input, the forest takes the estimated scan cardinality of each table that the query accesses; these cardinalities can be computed using an off-the-shelf cardinality estimator. The forest is trained as the final step in blueprint

optimization using run times that our query run time model predicts (the engine with the lowest predicted run time is the most preferred). Inference over the decision forest is fast and does not impose an undue overhead on the queries. We empirically evaluate the effectiveness and overhead of our routing policy in Section 5.4. We discuss additional practical details for query routing in Section 4.3.

4 BRAD'S BLUEPRINT PLANNER: DETAILS

In this section we outline the implementation details behind BRAD and provide additional details about BRAD's blueprint planner.

4.1 Realizing the Blueprint Planning Life Cycle

Figure 3 depicts BRAD's system architecture, which implements the blueprint planning life cycle using two components: (i) a front-end server responsible for interfacing with clients and operating the blueprint, and (ii) a blueprint planner that monitors the workload and selects new blueprints when appropriate. We describe the architecture by walking through the blueprint planning life cycle.

The life cycle begins at BRAD's front-end server (Figure 3 Ⓐ). Users submit SQL queries to the server, which get routed to a suitable engine for execution (Section 3.4). Crucially, to keep track of the executing workload, the front end (i) logs the queries it receives Ⓑ, and (ii) collects metrics about the workload (e.g., transaction latency, query latency). The blueprint planner monitors these metrics Ⓒ, alongside others it retrieves from the underlying engines (e.g., CPU utilization) and triggers blueprint optimization when they exceed or fall below specified thresholds Ⓓ (Section 4.6)

When starting blueprint optimization, BRAD first extracts the queries that ran during its planning window (a sliding window of a configurable length Ⓔ) from its workload log. These queries, along with dataset statistics (e.g., the sizes of the tables), are passed to the optimizer and represent the workload that BRAD uses when scoring candidate blueprints Ⓕ. BRAD's optimizer then searches over valid blueprints (Section 3.3), scores them (Section 4.2), and returns the best-scoring blueprint Ⓖ (Section 4.5). The planner then transitions the infrastructure to the chosen blueprint and passes it to the front end Ⓗ. The front end uses this blueprint until the next one is chosen, completing the blueprint planning life cycle.

4.2 Additional Blueprint Scoring Details

4.2.1 Query Run Time and Data Scanned. As discussed in Section 3.2, BRAD uses a graph neural network with a novel query featurization to predict a query's run time and the amount of data it scans. We now describe the featurization and model in more detail.

Query featurization. As discussed, existing run time predictors require the query's physical execution plan as input, which is not always available in BRAD's setting (Section 3.2). Thus, we design a new graph featurization approach to encode information, such as filters, joins, and group-bys, purely from a query's SQL. This logical featurization differentiates our GNN from existing models [50, 70, 71, 73, 100, 114]. Figure 4 shows an example procedure to extract such a query feature graph. It has five types of nodes, each with distinct node features (shown in Table Ⓓ in Figure 4) and edges representing the dependencies between these nodes. We parse the query's SQL to extract the tables, columns, predicates, and logical operations it involves and construct the feature graph in four steps.

First, we extract the tables and columns involved in the query. We show an example table A and its relevant columns A_1, A_2, A_3 in blue in Figure 4 Ⓐ. We create a table node for each table and a column node connecting to a table node for each column (Figure 4 Ⓑ). Second, we extract the operations on a single table, such as "scan", "aggregate", and "group-by", as highlighted in green in Figures 4 Ⓐ and 4 Ⓑ. Specifically, we create a predicate node for each filter predicate and connect the column nodes involved in this predicate to it. We extract the features of the predicate nodes using an approach similar to recent work [50, 100]. We connect these predicate nodes to their parent operation node. For operations without a predicate, such as "aggregate", we just connect them to the relevant column nodes. Third, we parse the operations involving multiple tables, e.g., "join", as highlighted in red. The children of each join operation node are the relevant scan operation nodes and the join predicate node extracted the same way as a filter predicate. We list the features for all node types in Figure 4 Ⓓ. Finally, as highlighted in yellow in Figure 4 Ⓑ, all the operation nodes are connected to the embedding node, which aggregates the overall information.

This generic query graph representation does not encode any information about a physical plan (e.g., join order or physical operators). The features we use can all be independently derived without relying on BRAD's underlying engines. We use a value of -1 for the aforementioned node features if the feature is unavailable.

Selectivity estimates. Our model also uses an estimate of the selectivity of each operation node. This is because the selectivity influences the chosen physical plan, and thus the query's run time as well. BRAD collects simple statistics about each table (e.g., histograms) using analysis pass over the underlying data. BRAD then uses the Selinger method [96] to make these estimates because of its simplicity and negligible overhead. The choice of estimator is orthogonal to our model; other methods are also applicable [81, 113].

Graph neural network. Inspired by the zero-shot run time predictor [50], we use a graph neural network to model interactions between nodes and propagate information through the feature graph. We construct one multi-layer perception (MLP) node encoder for each level that embeds the node features into a fixed-length vector. Then, we create another MLP for message passing through edges [42]. At each internal node, we sum its children's embedded vectors, concatenate them with its own vector, and feed the resulting vector to the MLP to get a new vector. The message passing stops at the last level when the "embedding node" has aggregated all of the query's information into a single vector. This vector is fed to two MLPs (Figure 4 Ⓒ) to predict the query's run time and amount of data scanned. The entire model is trained end-to-end.

Discussion. By using logical features and selectivity estimates, our model implicitly learns a query optimizer's behavior and makes accurate predictions for queries similar to our training queries (see Section 5.3). Our model may not work well in some cases where the testing queries are *significantly* different from our training queries. For example, suppose the model was only trained on short running queries joining small tables and the user then submits a long running query joining large tables. Then, the engine could possibly choose a join operator/order that it has never chosen before, and

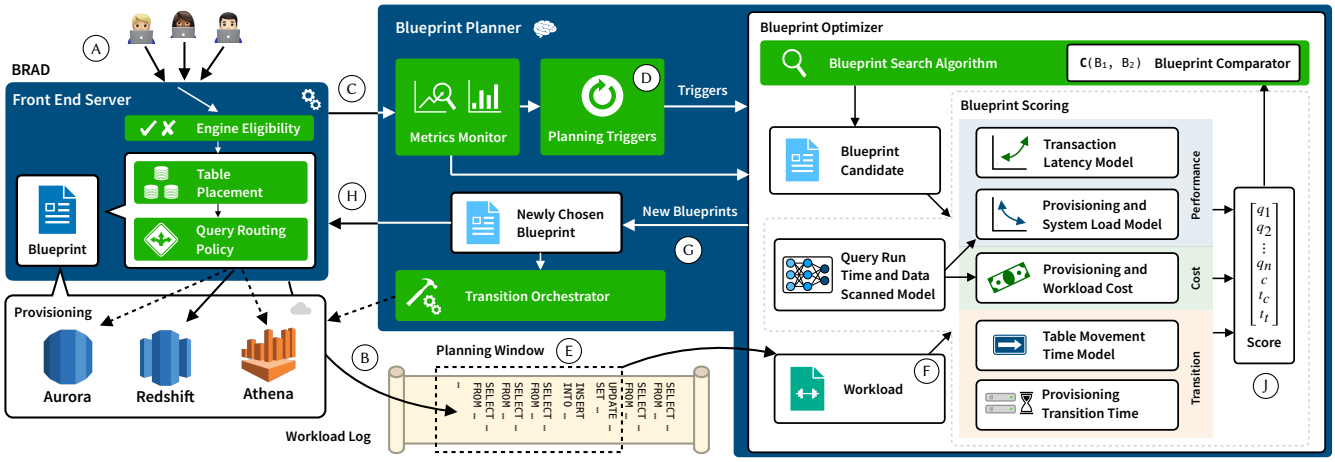

Figure 3: A detailed view of BRAD's architecture and its end-to-end blueprint planning life cycle.

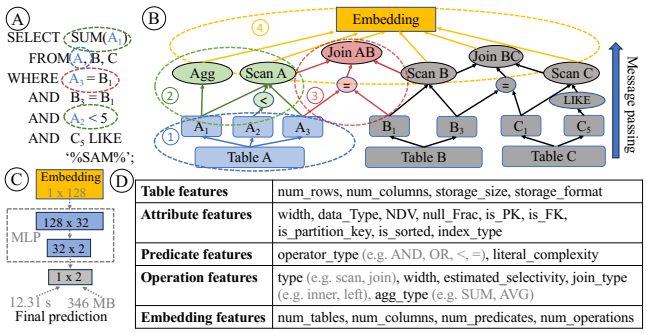

Table features	num_rows, num_columns, storage_size, storage_format
Attribute features	width, data_Type, NDV, null_Frac, is_PK, is_FK, is_partition_key, is_sorted, index_type
Predicate features	operator_type (e.g. AND, OR, <, =), literal_complexity
Operation features	type (e.g. scan, join), width, estimated_selectivity, join_type (e.g. inner, left), agg_type (e.g. SUM, AVG)
Embedding features	num_tables, num_columns, num_predicates, num_operations

Figure 4: The query featurization used by our model, which predicts a query's run time and the amount of data it scans.

our model (and possibly all existing run time models) would be unlikely to correctly predict that query's run time. However, recent work shows that in practical workload traces, a large portion of queries are repeating and "similar" to prior queries [112]. Thus, encountering a query that greatly impacts the selected plan in a way that was not captured in the training data should be rare.

4.2.2 Provisioning and System Load. Next, we describe the two analytical models we use to adjust our GNN's run time estimates for different provisionings and system load.

A query's run time consists of two components: the time spent running and the time it queues due to system load. Thus, BRAD models a query's complete run time R as $R(G, \rho) = P(G) + W(\rho)$, where G is the run time given by our GNN, $P(\cdot)$ is a model that accounts for the engine's provisioning, and $W(\cdot)$ is the time spent queuing depending on the system's utilization (load) ρ.

Compute resources. BRAD uses $P(G) = (c_1(b/d) + c_2)\,G$, which we derive from Amdahl's law [22]. We model the query's run time as having two parts: (i) one that will decrease (or increase) if the engine's provisioning is changed to have more (or less) compute resources, and (ii) a fixed part that will not change even with more resources. Here $c_1 G$ represents part (i) and is multiplied by b/d, which is the ratio between the resources available on the "base" and "destination" provisionings. The base is the provisioning on which the graph neural network was trained. The destination is the engine's provisioning in the candidate blueprint. The $c_2 G$ term is part (ii). BRAD uses the total number of vCPUs in the provisioning to represent the available compute. We learn one set of constants c_1 and c_2 for each engine (Aurora and Redshift) using least squares linear regression [26] on the query workload. This approach assumes that provisioning changes do not cause significant query plan changes that would affect the run time. Empirically, we find this model to be sufficient for blueprint planning (Section 5.2).

System load. BRAD models $W(\rho_r)$ using queuing theory, approximating an engine as an M/M/1 system [48]. We use $W(\rho_r) = -K\,(1-\rho_r)^{-1} \log\left(\rho_r^{-1}(1-q)\right)$ which models the q-th percentile queuing time on a system with utilization ρ_r [2]. K represents the mean processing time, which we estimate as the average $P(G)$ for all queries assigned to the engine. We approximate an engine's utilization using its measured CPU utilization. In our experiments, BRAD optimizes for a p90 latency constraint, so we use $q = 0.9$. We use this model as it provides a simple closed-form expression for wait time. Not all engines are M/M/1 systems; for example, the query arrival distribution may not be exponential and/or the engine may process queries in parallel. However, we empirically find that this simple model is sufficient for blueprint planning (Section 5.2).

Adjusting ρ_r. $W(\rho_r)$ relies on a representative ρ_r. BRAD cannot directly use CPU utilization because the candidate blueprint may use a different query routing as the current blueprint, thereby imposing different loads. Instead, we assume that a query's "load" is proportional to its run time. BRAD thus scales its observed CPU utilization by a factor: the sum of the run times of the queries routed to the engine in the candidate blueprint divided by the sum of the run times that actually ran on the engine in the last planning window. If no queries ran on the engine in the previous planning window (e.g., the engine was paused), BRAD scales the query's predicted run times to a load value using a learned proportionality constant.

4.2.3 Transaction Latency. BRAD estimates transactional latency on a candidate blueprint to ensure it provisions Aurora appropriately for the transactional load it experiences. In general, estimating a transaction's run time is a hard problem due to the many factors

that can influence its latency (e.g., lock contention, buffer pool state, etc.) [79]. We make a simplifying assumption that the transactional workload is uncontended and consists of TPC-C-like [101] indexed point reads and writes made interactively over the network. For this setting, we use an analytical function that models a transaction's latency as a function of system utilization: $R(\rho_t) = a/(M - \rho_t) + b$. Here, $R(\rho_t)$ is the overall transactional latency. a, b, and M are workload-specific learned constants. $\rho_t \in [0, 1]$ is the system utilization; we require that $M > \rho_t$. We use CPU utilization as a simple proxy metric for ρ_t. This model captures that transaction latency increases rapidly as ρ_t approaches M, like it would on an overloaded system [48]. We derive this model empirically; it is loosely based on the queuing theory equations for an M/M/1 system [48].

Adjusting ρ_t. The candidate blueprint's ρ_t may not be the same as the measured ρ_t on the current blueprint (e.g., due to a changed provisioning and/or query routing). BRAD applies two scaling factors to compensate. First, it multiplies ρ_t by the ratio of vCPUs between the candidate blueprint's provisioning and the current blueprint's provisioning. Second, it applies the same query load scaling factor mentioned in Section 4.2.2 to account for query movement.

4.2.4 Operating Cost and Transitions. A blueprint's cost comprises provisioning costs, Athena query costs, and storage costs. For Aurora and Redshift, BRAD uses AWS' on-demand instance pricing [13, 16]. Currently BRAD only considers Aurora I/O optimized instances, which do not bill I/O usage [8]. Since Athena bills by the amount of data scanned, BRAD uses its data scanned predictions (Section 4.2.1) along with Athena's scan pricing [11]. For storage costs, BRAD models a table's size as $k|T|$ where $|T|$ is the number of rows in the table and k is an engine and table-specific constant. To compare blueprints, BRAD normalizes costs to be in $/hour.

Table movement. BRAD currently moves tables via S3 (i.e., export to S3 and import from S3). It estimates the movement time as $S/k_e + S/k_i$, where S is the physical size of the table and k_e and k_i are empirically measured export and import rates. These rates are engine-specific but independent of the engine provisioning, which we confirmed empirically. AWS does not charge for S3 transfers between AWS services, so BRAD does not incur movement costs.

Aurora provisioning time. BRAD estimates Aurora's provisioning time as the number of instance changes multiplied by a fixed amount of time (we empirically measured 5 minutes). Removing replicas is considered to take no time since BRAD does not need to wait for the completion of removal to start using the next blueprint.

Redshift provisioning time. The time it takes to complete a Redshift provisioning change depends on whether it can be done using an elastic resize or not [9]. For elastic resizes, BRAD uses AWS' published estimate of 15 minutes [9]. BRAD estimates classic resizes to take $|R|/k$ where $|R|$ is the physical size of the data in the Redshift cluster. We empirically observed k to be approximately 18 megabytes per second. Pausing Redshift is also considered to take no time for the same reason as Aurora replica removals.

4.3 Additional Query Routing Considerations

Upon receiving a query, BRAD selects a suitable engine in two steps: (i) determine the set of engines that are *able* to execute the query, and then (ii) select the most suitable (e.g., best performing) engine from this set. In this section, we describe step (i). For (ii), BRAD uses the online routing policy described in Section 3.4.

In BRAD, an engine's eligibility to run a query depends on the *table placement* and its *functionality support*. Table placement is the set of engines that hold a copy of a table and is governed by the blueprint; BRAD currently only routes a query to an engine if it has a copy of every table the query accesses. BRAD parses the SQL query to extract the tables it references and compares them against the blueprint's table placement. During blueprint planning, BRAD ensures that all tables are placed together on at least one engine to so that it can always run a query that joins any subset of tables.

A query may also use specialized functionality only available on a subset of the engines (e.g., vector similarity search [64, 90]). By taking functionality into account, BRAD ensures that it only selects engines that can run the query. Automatically determining the "specialized functionality" that a query uses is something we leave to future work. BRAD currently uses keyword matching against pre-specified keyword lists to determine if a query uses such functionality (e.g., the presence of the <=> operator would imply that the query uses vector similarity search).

4.4 Additional Blueprint Search Details

Algorithm 1 contains the pseudocode for BRAD's greedy beam blueprint search algorithm, which we outline in Section 3.3. The intuition for using a top-k beam search instead of a naïve greedy search lies in the nature of the search space. At the beginning of the search, assigning queries to some engines may be better (e.g., prefer Athena, which is pay-per-query, instead of Redshift where you pay for provisioning). But after assigning more queries, this trade-off changes (e.g., there are enough queries to justify running Redshift). Keeping a set of promising candidates helps BRAD balance these changing trade-offs. We search over nearby provisionings because BRAD handles gradual workload changes; the next best provisioning is likely to be near the current provisioning.

Discussion. Beam search works well empirically in our setting for two reasons. First, BRAD uses a large beam ($k = 100$), which helps prevent some promising candidates from being eliminated too early. Second, the queries in our workload have a skewed arrival frequency (i.e., some queries arrive more frequently than others). This property was also observed by our industrial partners in their real-world workloads [112]. As a result, BRAD processes queries in decreasing order of arrival frequency. Along with using a large beam, this processing strategy makes it more likely for "important" (i.e., frequently occurring) queries to be assigned to the best engine.

Analysis. Let m be the number of engines considered, q be the number of queries in the workload, and p be the number of distinct provisionings considered. This algorithm considers $O(kmqp)$ candidate blueprints. Currently, BRAD has $m = 3$ and uses $k = 100$. We evaluate our algorithm empirically in Section 5.5.

4.5 Blueprint Comparator: Minimizing Cost

As discussed in Section 3.1, end-users need to define a comparator function, which imposes an ordering on blueprint vector scores. This comparator is how users convey their infrastructure design goals to BRAD. A common goal is to minimize an infrastructure's operating costs while maintaining a service level objective (SLO)

(e.g., p90 query latency should be under 30 seconds). We use this design goal when evaluating BRAD in Section 5. We now describe how this goal is implemented as a comparator.

Given two blueprints (B_1, B_2), the general idea is to map their vector scores to scalar costs (W_1, W_2); the candidate with the lower cost is considered better. A simple mapping would be to use the blueprint's operating cost when the predicted query latency falls under the desired latency constraint and an infinite cost otherwise (to indicate infeasibility). However, this mapping does not consider the time and cost of transitioning to the candidate. Instead, our approach is to weigh the cost of operating each blueprint using the transition time T_T and a user-defined "benefit period" T_B. This period represents how long the user expects the workload to "benefit" from the new blueprint. Concretely, we use

$$W = P^{\gamma} C_0 T_T + C_T + C T_B \qquad P = 1 + \max\left(t/t_{\text{SLO}}, q/q_{\text{SLO}}\right)$$

C_0 represents the current blueprint's operating cost, C is the candidate blueprint's predicted operating cost and C_T is the transition cost. $P \in [1, \infty)$ is a penalty multiplier that grows as the current blueprint approaches and exceeds the performance constraints (e.g., because the workload changes). γ is a user-chosen weight (we use $\gamma = 2$). t and q represent the transaction and analytical latency measured on the current blueprint; t_{SLO} and q_{SLO} represent the user-specified performance constraints for these values. Users can declare multiple such constraints (e.g., for different queries).

When the current blueprint exceeds the performance constraints, the first term in the equation on the left will dominate. Thus BRAD will prefer candidate blueprints that are faster to transition to. Otherwise, BRAD weighs the operating costs by the time spent transitioning versus running the new blueprint. This means BRAD will still consider blueprints requiring very expensive transitions (high T_T) but will only select them if their benefit is large enough (low C during T_B). If a candidate blueprint has a predicted transactional or analytical latency greater than t_{SLO} or q_{SLO}, we just assign an infinite cost. If all of the candidates are infeasible, BRAD will ask the user to change their constraints.

4.6 Triggering Blueprint Optimization

BRAD periodically checks a set of *triggers* to decide when to initiate blueprint optimization. BRAD initiates blueprint optimization if one of them fires. Concretely, BRAD uses the following triggers:

- **Aurora / Redshift CPU utilization.** BRAD triggers blueprint optimization if they consistently violate preset thresholds. (e.g., exceeding 85% or falling below 15% for 10 minutes or more).
- **Transaction and query latency.** When optimizing for cost under a performance SLO, BRAD will trigger re-optimization if these latencies consistently exceed the user's SLOs.
- **Recent provisioning change.** If BRAD selects a new blueprint with a provisioning change, it will trigger re-optimization after a fixed period of time to ensure performance is as expected. This is because BRAD makes conservative provisioning decisions to avoid selecting blueprints that will violate the user's SLOs. Re-optimizing after the new blueprint takes effect gives BRAD an opportunity to revisit its choice after observing the workload on the new blueprint.

Algorithm 1 BRAD's greedy beam blueprint search algorithm.

Input: B_0: Current blueprint, W: Workload,
Score$(\cdot,\cdot,\cdot)$: Blueprint scoring function
Output: B^*: Best found blueprint

```
function DoSearch(P: Provisioning)
    Sort queries in W in decreasing order of arrival probability
    and then largest predicted speedup across engines
    ▷ Initial blueprint with provisioning P and no routed queries ◁
    T ← [(B(P, ∅), Score(B(P, ∅), B_0, W))]
    for all queries q in W do
        T' ← [ ]
        for all blueprints B in T do
            for all engines e in {Aurora, Athena, Redshift} do
                B' ← (B ∪ {q → e})    ▷ Route query q to e in B'
                if B' is valid then
                    T' ← T' add (B', Score(B', B_0, W))
        ▷ Note that T' is implemented as a top-k heap. ◁
        T ← T' truncated to the top-k candidates
    return Best candidate in T
B* ← None
for all provisionings P near the provisioning in B_0 do
    B ← DoSearch(P)
    if B is better than B* then
        B* ← B
output B*
```

4.7 Discussion

Blueprint planning practicality. Our blueprint planning framework has three practical benefits. First, blueprints and their scores are *human-interpretable*, making it easy for data engineers to inspect BRAD-chosen designs. Second, blueprints provide a useful abstraction for realizing cloud infrastructure designs. Conceptually, they can be "compiled down" into infrastructure-as-code manifests (e.g., CloudFormation [18]) for deployment. Finally, blueprints are generalizable to other cloud infrastructure design problems that involve cost/performance-based resource provisioning, task scheduling, and adaptation under changing conditions. Some example use cases include resource configuration selection for Ray [77] programs, designing long-lived infrastructures used by Sky intercloud brokers [30], or assembling a model serving service [93].

Adding engines to BRAD. BRAD can support additional engines beyond Aurora, Redshift, and Athena. For an engine to be included in BRAD, it must (i) support relational data, (ii) have a SQL-based query interface, (iii) expose system metrics (e.g., CPU utilization), (iv) use a deterministic query planner, and (v) have deterministic operational costs. Practically, the engine should also have management APIs that allow BRAD to programmatically alter its allocated resources (i.e., provisioning) to deploy blueprints. By (iv), we mean that the query planner must pick the same physical plan for the same query if it has the same dataset statistics (e.g., estimated scan selectivity) (see Section 4.2.1). Finally by (v), we mean that the cost of running the engine in the cloud must be a deterministic function of the engine's physical configuration (e.g., its provisioning and the size of its data) and the user's workload. For example, the engine's

operating cost cannot depend on external factors that BRAD cannot directly observe (e.g., resource demands from other cloud users).

5 EVALUATION

In our evaluation, we seek to answer the following questions:

- How effective is BRAD at optimizing a data infrastructure for cost when compared to serverless autoscaling systems? (Section 5.2)
- How accurate are the models that BRAD uses to score its candidate blueprints and how well do they generalize? (Section 5.3)
- How effective is BRAD's query routing and how much overhead does BRAD add to query execution? (Section 5.4)
- How effective is BRAD's blueprint search algorithm? (Section 5.5)
- How sensitive is BRAD's blueprint planner to model errors? (Section 5.6)

Across five workload scenarios, we find that BRAD selects designs that meet performance targets while outperforming a serverless Aurora and Redshift infrastructure and a serverless HTAP system (where comparable) on cost by up to 13× and 4.6× respectively.

Overall implementation. We implemented BRAD in Python using approximately 30k lines of code. Although BRAD currently uses AWS services, the concepts underlying blueprint planning and our scoring models are general and applicable to other cloud providers.

5.1 Workload and Experimental Setup

We evaluate BRAD on a new workload that models the data processing needs of a fictitious movie theater company called *QuickFlix*.

Why create a new workload? BRAD automates the design of multi-engine cloud data infrastructures serving diverse transactional and analytical workloads. Thus, we need a realistic and diverse workload that warrants multiple specialized engines. To our knowledge, no such public workloads exist. The TPC [102, 103] and HATtrick benchmarks [76] are entirely synthetic. IMDB JOB [63] and STATS CEB [47] use real-world datasets and queries, but only contain OLAP queries as they are for evaluating query optimizers. Snowset [108] has statistics about real OLAP workloads, but no data or queries. Our workload addresses these limitations: it (i) contains transactions and diverse analytical queries, (ii) adapts a real-world dataset, and (iii) mimics Snowset statistics where possible.

Dataset. We use an adapted version of the IMDB dataset [63], which is based on real-world data. As the original IMDB dataset is small (3 GB), we create a larger dataset by replicating each tuple in the dataset's major tables 30 times. Then, we assign new values for each replicated primary key and re-assign these values to their corresponding foreign keys. This approach preserves the dataset's attribute correlations, skew, and join-key distributions. We additionally add three synthetic tables representing movie theaters, movie showings, and ticket orders to capture the company's transactional needs. The final uncompressed dataset is 160 GB.

Analytical queries. Our workload consists of two classes of analytical queries: (i) recurring queries (e.g., used for QuickFlix's dashboards and interactive internal tools), and (ii) complex ad-hoc analytical queries (e.g., representing exploratory data analysis). The recurring queries consist of single table scans and two table inner joins, both with predicates. The complex ad-hoc queries are randomly generated to resemble the IMDB JOB queries. They span hundreds of distinct templates that join 4 to 15 tables with complex filter predicates. Of the unique queries in our workload, 80% are recurring and 20% are ad-hoc; we chose this split to match what our industry partners have observed in their production workloads.

Transactions. We use 3 transaction types: (i) Purchase, (ii) AddShowing, and (iii) UpdateMovie. Purchase looks up a theater, selects a showing, inserts a ticket order, and updates the showing's seat count. AddShowing looks up a theater and movie and inserts a new showing record. UpdateMovie selects a movie from the "title" table and edits the corresponding note column in the "movie_info" and "aka_title" tables. We run these transactions with a breakdown of 70% Purchase, 20% AddShowing, and 10% UpdateMovie.

Baselines. We compare BRAD against (i) an infrastructure using serverless Aurora for all transactions and serverless Redshift for analytics, and (ii) System H, a popular open-source serverless HTAP database. Note that these comparisons are not perfectly fair as these systems provide different guarantees. We select them because they represent existing industry-standard infrastructure solutions that provide the same hands-off autoscaling experience.

Metrics. We record three metrics: (i) transaction latency, (ii) analytical query latency, and (iii) monthly operating cost. In our experiments, BRAD optimizes a data infrastructure to minimize operating cost while ensuring that p90 transaction latency remains under 30 milliseconds and p90 query latency remains under 30 seconds.

Operating cost calculations. We compute BRAD's operating cost using the on-demand instance hourly cost scaled to 30 days. For queries running on Athena, we compute their cost using the reported bytes scanned. We project this value into a monthly cost by assuming that the query repeats at the same observed rate. We include storage costs for the tables placed on Aurora and Athena (S3). For serverless Aurora and Redshift, we use the ACU and RPU values reported by AWS during the workload and scale them to monthly costs. For System H, we use the cost reported by the vendor.

Considered instance types. For Aurora, BRAD currently only considers Graviton-based instances [6] and I/O optimized cluster configurations (i.e., I/O costs are included in the hourly provisioning cost) [8]. We leave the consideration of different instance hardware types (e.g., r6g vs. r6i instances) to future work. For Redshift, BRAD considers the dc2 and ra3 family of instance types.

5.2 Optimizing a Data Infrastructure

We have BRAD optimize a data infrastructure for cost under a performance constraint in five scenarios faced by QuickFlix:

(1) Scaling down an over-provisioned infrastructure.
(2) Scaling engines due to increased load.
(3) Maintaining support for specialized functionality.
(4) Adjusting to user-changed constraints.
(5) Workload intensity variations during a day.

5.2.1 Scaling Down an Over-Provisioned Infrastructure. QuickFlix has been struggling with their data infrastructure. After learning about BRAD, they adopt it to free up their data engineers. QuickFlix deploys BRAD on their infrastructure, consisting of two Aurora db.r6g.xlarge instances (primary and read replica) and two dc2.large Redshift nodes. Following conventional wisdom, they use Redshift

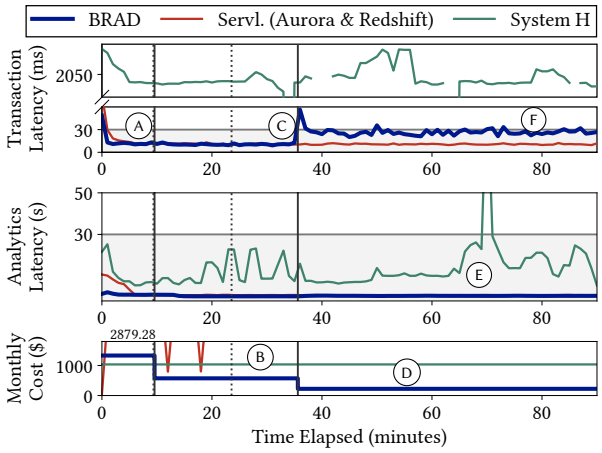

Figure 5: BRAD reduces cost while maintaining p90 latency constraints (shaded region). The dotted (solid) vertical lines indicate when a new blueprint is chosen (takes effect).

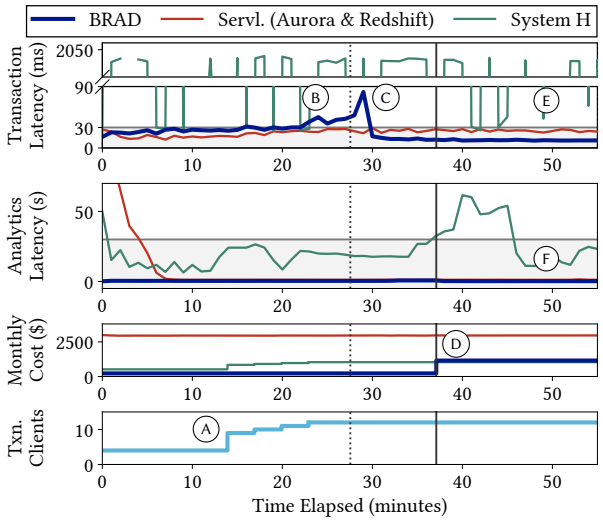

Figure 6: As transactional load increases, BRAD switches to a larger Aurora instance and also removes the read replica.

for analytical queries and Aurora for transactions. Figure 5 shows workload performance and the monthly operating cost over time. The shaded area indicates QuickFlix's performance constraints: 30 ms p90 transaction latency and 30 s p90 analytics latency.

Soon after starting, BRAD triggers blueprint optimization (A) because it detects a low Redshift CPU utilization. BRAD removes the Aurora read replica, shifts the entire analytical workload onto Aurora, and pauses Redshift. BRAD makes these changes to reduce cost (B), as it correctly predicts that Aurora is sufficient to handle the workload. After observing the workload on this new blueprint, BRAD then correctly predicts that Aurora can support the workload with a smaller (cheaper) instance type and thus downscales Aurora to two db.t4g.medium instances to further reduce cost (D). The momentary spike in p90 transactional latency is due to the Aurora primary failover that occurs when changing instance types (C). The chosen blueprint meets QuickFlix's performance constraints (E) (F). From these results, we draw the following two conclusions.

BRAD reduces operating cost by 6.0× over its starting provisioning and by 4.6× over System H, the next best baseline. Serverless Aurora and Redshift is 13× more expensive than BRAD's because serverless Redshift has a large minimum size, making it cost-ineffective on this workload. Although System H is only 4.6× more expensive than BRAD, its p90 transaction latency is nearly 100× higher than the other baseline. We hypothesize that this is due to System H's internal replication on writes.

BRAD shifts workloads across engines to reduce cost, differentiating it from static multi-engine autoscaling infrastructures. BRAD correctly predicts that Aurora can support the analytical workload, enabling it to pause Redshift to reduce cost. This decision would never be considered in static autoscaling infrastructures, such as our serverless Aurora and Redshift baseline, since they only scale to respond to system load while keeping the workload assignment fixed (i.e., analytics always run on Redshift).

5.2.2 Scaling Engines Due to Increased Load. As QuickFlix grows, their transactional load increases. Figure 6 shows how BRAD handles this change; we plot the same metrics as before and include the number of transactional clients over time (A). We begin with the same blueprint as the end of the previous scenario. After a few minutes, BRAD notices that the transaction p90 latency is exceeding QuickFlix's 30 ms ceiling (B) and triggers blueprint optimization. BRAD chooses to scale up Aurora to a single db.r6g.xlarge instance as it correctly predicts that a single Aurora instance can support both the increased transactional load and the running analytical queries. The spike in p90 transactional latency (C) is when the Aurora primary failover occurs. On the new blueprint, the transactional p90 latency falls under the latency ceiling (E); the analytical queries also continue to complete under the 30 s ceiling (F). Again, the increased System H analytics latency might be due to a combination of its internal physical autoscaling and storage writes.

This shows that BRAD reacts to transactional load to maintain latency constraints. The blueprint that BRAD selects is 2.6× cheaper than the serverless Aurora and Redshift baseline (D) but up to 1.1× more expensive than System H. Serverless Aurora and Redshift is more expensive due to Redshift's large minimum size.

5.2.3 Maintaining Support for Specialized Functionality. To increase engagement, QuickFlix decides to deploy a new feature that recommends movies similar to the ones shown in their theaters. To make recommendations, they use *vector similarity search* [64, 90] queries that find movies with title embeddings that are closest to a given movie's title embedding. QuickFlix deploys this feature on their existing infrastructure; since similarity search is only supported on Aurora, they place the relevant tables on Aurora and run all other queries that access these tables on Aurora as well. They use two Aurora db.r6g.2xlarge instances and two dc2.large Redshift nodes.

Figure 7 shows performance and infrastructure cost over time. The dashed lines are from before BRAD deploys its first optimized blueprint. Crucially, System H *cannot* run this workload because it does not support vector similarity search (A). BRAD notices that the analytical p90 latency exceeds QuickFlix's constraint of 30 s (B) and initiates blueprint optimization. BRAD selects a blueprint that shifts the non-vector similarity search queries onto Redshift

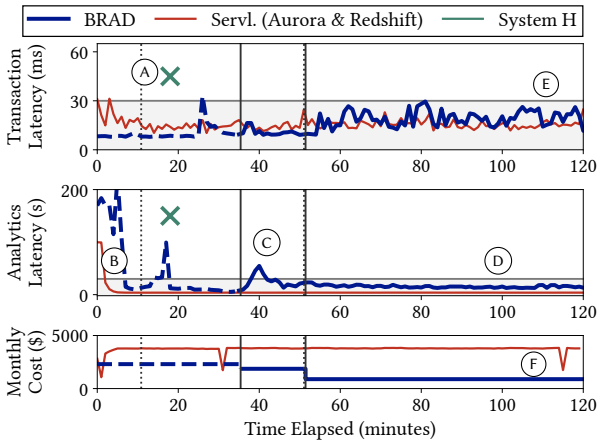

Figure 7: BRAD runs vector similarity search on Aurora and shifts other queries to Redshift for performance. System H does not support vector similarity search.

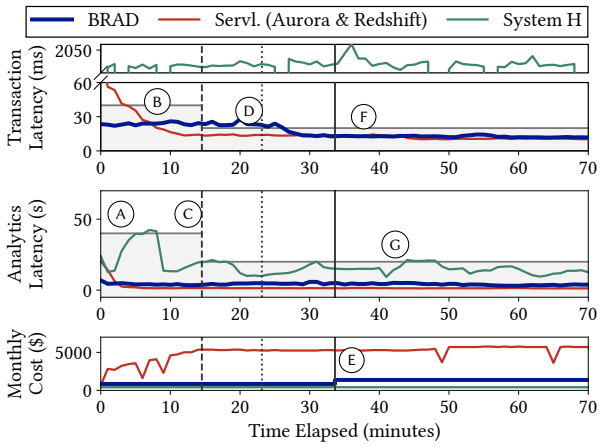

Figure 8: After the user changes their SLO constraints, BRAD selects a new blueprint to meet the new constraints.

(replicating the tables they access onto Redshift) while keeping the vector similarity search queries on Aurora. It also correctly predicts that it can downscale Aurora (to a db.r6g.xlarge instance) to save cost, as much of Aurora's former query load was moved onto Redshift. After making this change, the workload's performance falls within the user's performance constraints (D) (E). The momentary spike in analytics latency at the 40 minute mark (C) is due to a cold Redshift cache when BRAD first moves queries onto Redshift. The serverless Aurora and Redshift design is 2.4× more expensive (F) because of the large Redshift minimum size and because Aurora has scaled up to support the new similarity search queries.

This scenario shows how BRAD is fundamentally different from single-system (e.g., HTAP) solutions like System H. When using a single system to run a diverse data workload, you can always run into situations where you want to use a feature that does not exist on your system of choice. In contrast, in the BRAD architecture, one can (in concept) always *incorporate* a system with the required functionality into the underlying infrastructure.

5.2.4 Adjusting to Changed Constraints. As QuickFlix's business changes, they revise their performance constraints; Figure 8 shows how BRAD adapts to their new needs. QuickFlix initially uses a p90 transaction and query SLO of 40 ms and 40 s respectively (A) (B). BRAD starts with one db.r6g.xlarge Aurora instance and two Redshift dc2.large nodes. Later, QuickFlix lowers their transaction and query SLOs to 20 ms and 20 s respectively (the change happens at the dashed line in Figure 8 (C)). This SLO change causes the transaction latency to exceed QuickFlix's constraints. Thus, BRAD scales up Aurora to one db.r6g.2xlarge instance (D) and leaves Redshift as-is. This change increases the operating cost (E) as BRAD switches to a larger Aurora instance. After BRAD finishes transitioning the infrastructure, the transaction latency falls under the new 20 ms p90 constraint (F). The query latency constraint also remains under the new 20 s p90 constraint (G). BRAD's operating costs are 4.2× lower than the serverless Aurora and Redshift baseline. This is because serverless Redshift has a large minimum size. While System H ends with a 4.4× lower monthly operating cost than BRAD, it does not meet the 20 ms transaction latency constraint (its transactions have a latency around 2 seconds). We again think that this elevated latency is due to System H's internal replication on writes. This scenario shows that BRAD adapts to changes to a user's constraints.

5.2.5 Workload Intensity Variations During a Day. Finally, we run BRAD on a workload representing a full day at QuickFlix. For practical reasons, we scale the actual workload to 12 hours. Figure 9 shows performance and cost over the day. We use the workload and dataset from Section 5.1 adapted to mimic the Snowset trace [108]. Concretely, we run queries with a run time distribution similar to the Snowset trace and vary the number of clients issuing queries and transactions to mimic the diurnal pattern observed in Snowset (a peak near the middle of the day, Figure 9 (F)).

Initially, the workload is light. BRAD uses a blueprint with four dc2.large Redshift nodes and two Aurora db.t4g.medium instances, which is 2.5× cheaper than the serverless Aurora and Redshift baseline (A). As the workload intensity increases, BRAD detects the increases in latency (B) (C) and triggers blueprint optimization, ultimately scaling Redshift up to 16 nodes and Aurora to one db.r6g.xlarge instance at the peak. The serverless Aurora and Redshift baseline also scales up, but it does not consistently meet the analytics performance target of 30 s (D), despite being 2.1× more expensive than BRAD's design (E) at the workload peak. System H maintains the p90 analytics latency SLO throughout the workload, but its transactional latency is again almost two orders of magnitude higher than the other systems (we hypothesize for the same reasons as discussed earlier). The brief analytics latency spikes are due to Redshift resizes, which force clients to reconnect.

This result shows that BRAD effectively responds to load variations during the day. Over the day, BRAD maintains performance targets while reducing cumulative cost by 1.7× compared to the serverless Aurora and Redshift baseline.

5.3 Scoring: Model Accuracy and Generalization

We next examine the test accuracy of our predictive models and their generalizability across common workload shifts.

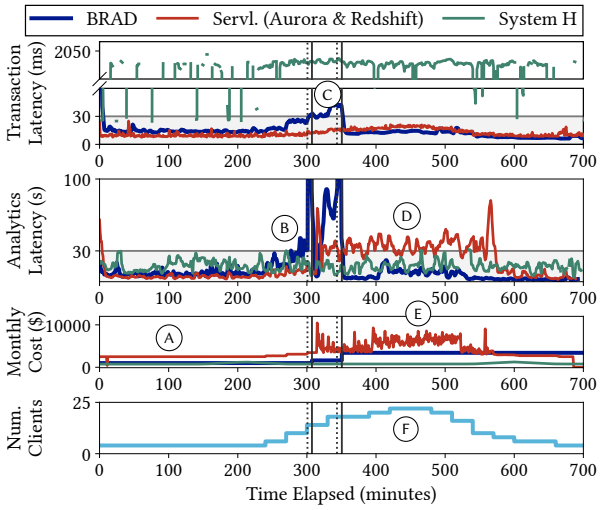

Figure 9: BRAD optimizes for load variations during the day.

Table 1: Median test Q error of our blueprint scoring models.

Prediction Target	Aurora	Redshift	Athena
Query Run Time	1.5769	1.6539	1.3427
Data Accessed	–	–	1.2614
Run Time on Different Provisioning	1.6718	1.6824	–
Transaction Latency	1.2030	–	–

5.3.1 Model Accuracy. Table 1 shows the median test Q error of our models for each engine. Q error is $Q(p, a) = \max(p/a, a/p)$, where p refers to the predicted value and a to the actual value. Lower is better; 1 is the best possible score. We train each run time and data accessed model using approximately 8000 queries, validate on 2000 queries, and test on 125 unseen queries. Our query dataset consists of over 1000 unique join templates. The models for Athena perform better than Aurora and Redshift because the run time distribution of Athena queries has a lower variance. We test our provisioning and transaction latency models on an unseen provisioning that is larger (i.e., has more resources) than all the training provisionings. Overall, we find that our models' prediction accuracy is sufficient for BRAD to design effective infrastructures (Section 5.2).

5.3.2 Generalizability. We evaluate our query run time model's generalizability on three workload shifts: (i) unseen join templates, (ii) adding a new table to the dataset, and (iii) a larger dataset size.

Unseen join templates. We train our run time models on queries with less than 5 joins. We then test the model's predictions on queries with 5, 6, and ≥ 7 joins. Figure 10 shows each model's median test Q error compared with (i) a model trained on all the join templates ("full"), and (ii) a naïve linear model that scales the cost returned by the engine's query optimizer to a run time. We label the percentage difference from the model trained on the full dataset. Our model generalizes across unseen join templates with a Q error of at most 20% above the model trained on the full dataset. Our model still performs much better than the naïve linear model, which has a Q error of at least 4.6. Since Athena's optimizer does not provide a query cost, we do not include a linear model result.

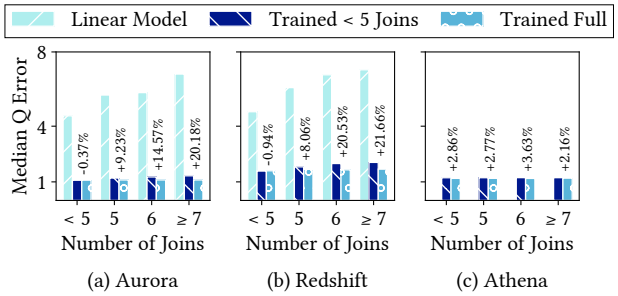

Figure 10: BRAD generalizes to unseen join templates.

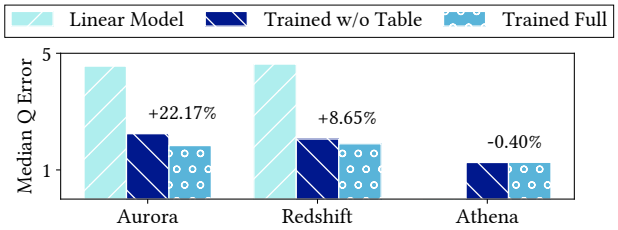

Figure 11: BRAD generalizes to an added table.

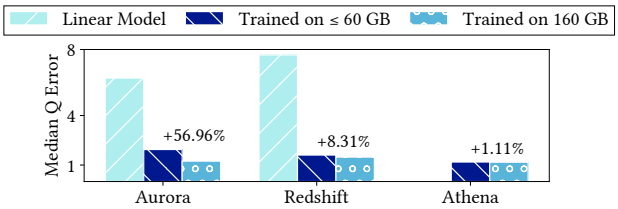

Figure 12: BRAD generalizes to an increased dataset size.

Added table. We train our run time models on queries that do not access the "person_info" table and test them only on queries that access the "person_info" table. This table is around 10 GB (the second largest table in the dataset); the overall dataset size is 160 GB. Figure 11 shows our results using the same baselines and notation as Figure 10. Our model generalizes to an added table with a Q error at most 22% above the model trained on the full dataset. The linear model still does poorly, with a Q error of 4.6.

Increased dataset size. Finally, we evaluate our run time model's generalizability to larger datasets. We train our models using queries executed on a 3 GB, 20 GB, 40 GB, and 60 GB version of our workload dataset. Then we test the models on the original 160 GB dataset. Figure 12 shows that our model generalizes to the larger dataset with a Q error of 1.1%, 8.3%, and 57% above a model specifically trained on the 160 GB dataset on Athena, Redshift, and Aurora respectively. The Aurora model has a higher error due to a change in caching behavior that occurs beyond 60 GB. The linear model has a Q error of 6.3 and 7.7 on Aurora and Redshift respectively. Overall, these results are sufficient for BRAD since an increase from 60 GB to 160 GB would likely happen over a longer period of time, allowing for BRAD to update its models given newly observed data.

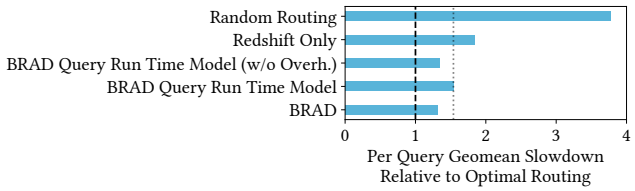

Figure 13: BRAD's routing quality is on par with its query run time model but without the inference overhead.

5.4 Query Routing Quality and Overhead

Next, we evaluate BRAD's query routing (Section 3.4) against four baselines: (i) selecting an engine randomly, (ii) routing all queries to Redshift, (iii) routing using the BRAD run time model (Section 4.2.1) but excluding its inference overhead, and (iv) routing using the BRAD run time model. We use 125 queries from our workload; 80% of the queries represent recurring queries and 20% are complex ad-hoc queries. We report the geomean slowdown over optimal per routing decision (i.e., lower is better, 1.0× is optimal). That is, for each query, we divide its run time on the chosen engine over its run time on the fastest engine, and take the geomean across queries.

Figure 13 shows our results. BRAD performs the best with a geomean slowdown of 1.31×, comparable to using the run time model and excluding its inference overhead (1.34×). Routing by actually using the run time model (i.e. including its inference overhead) has a geomean slowdown of 1.54×. These results highlight why we do not directly use our run time model for routing; it imposes a high overhead on the query's critical path (up to 115 ms), negatively affecting the routing performance. They also show that routing needs an intelligent strategy; the Random (3.78×) and Redshift Only (1.85×) strategies both perform worse than BRAD.

Overall query processing overhead. Similar to common database proxies [19, 24, 29], BRAD imposes some overhead. We measured a median overhead of 2.46 ms per complete transaction (these are interactive multi-statement transactions) and around 10 ms per analytical query. These could be further reduced, as BRAD is now implemented in Python, but we believe they are reasonable given that BRAD operates in the cloud, serving remote clients.

5.5 Blueprint Search Effectiveness

We next evaluate the effectiveness of BRAD's blueprint search algorithm. To compare the search algorithms, we report the final scalar score computed by BRAD's optimizer (Section 4.5). All baselines use the same scoring models and optimize for the same workload; they only differ in how they search for candidate blueprints.

We compare against three baselines: (i) uniform random sampling, (ii) naïve greedy, and (iii) exhaustive search. In uniform random sampling, each query is mapped to an engine that is chosen uniformly at random. We repeat this process to sample 10,000 blueprints and select the best one. In naïve greedy, each query is mapped to the engine with the lowest predicted run time (Section 4.2.1) without consideration of any other queries. Finally, exhaustive search looks through all possible mappings and therefore has a run time that is exponential in the number of queries. To be tractable, we use only 12 randomly chosen queries from the IMDB workload, comprising a search space of 530,000 candidates.

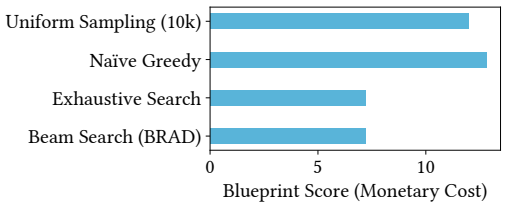

Figure 14: BRAD's blueprint planner compared to baselines.

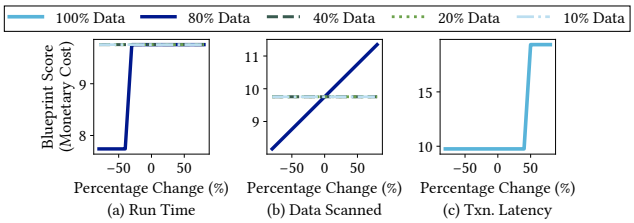

Figure 15: BRAD's planner is robust to prediction errors.

We compare the algorithms on the scale down scenario from Section 5.2.1. Figure 14 shows that BRAD's beam search finds a blueprint as good as exhaustive search. The score represents a monetary cost, and so lower is better. BRAD's blueprint's score is significantly lower than the blueprints selected through uniform sampling and the naïve greedy approach. This result indicates that BRAD's beam strategy is effective at finding optimized blueprints.

5.6 Blueprint Planner Sensitivity

Finally, we examine our blueprint planner's sensitivity to prediction errors. We inject errors into BRAD's predictions during blueprint planning and record the selected blueprint's score. Concretely, we select a random subset of the predicted values (query run time, data scanned, and transaction latency) and increase (or decrease) their predicted values by a percentage. We use subsets that include 10%, 20%, 40%, and 80% of the predictions. We run the blueprint planner on the scale down scenario from Section 5.2.1. Figure 15 shows our results. The x-axis is the amount of injected error, which we vary from −80% to +80%. The y-axis is the blueprint's scalar score. We study the effects of prediction error on query run time, data scanned, and transaction latency.

Figure 15(a) shows our results for query run time. There is no change in the selected blueprint even when up to 40% of the predicted query run times have injected errors of ±80%. When 80% of the predictions have injected errors of more than −40%, BRAD selects a different (cheaper) blueprint as it predicts that the cheaper blueprint can meet the performance constraints.

Figure 15(b) shows results for a query's predicted data scanned. Similar to run times, there is no change to the chosen blueprint when up to 40% of the predictions have injected errors. At 80%, BRAD chooses to route queries onto Athena. Thus the blueprint's monetary cost varies linearly with respect to the injected error.

Figure 15(c) shows results for transaction latency. Note that we inject errors into 100% of the data since BRAD makes just one latency prediction. Here, an injected error of +50% causes BRAD

to select a larger Aurora instance (to meet the latency constraint), which increases the blueprint score (monetary cost).

Overall, our results indicate that BRAD's planner is robust to prediction errors; more than 40% of the predictions need to have an error of more than ±50% for BRAD to choose a different blueprint. The intuition is that blueprints represent coarse-grained design decisions, and thus are more tolerant to prediction errors.

6 RELATED WORK

Instance-optimized, self-driving, and auto-tuning systems. Recent work has proposed techniques to automatically (i) adapt data systems to the workload [1, 35, 36, 57–59, 61, 70–72, 80, 89, 115, 116], (ii) manage complex systems [68, 78, 86–88, 95] (iii) adapt cloud database instance sizing [82–84, 106], and (iv) tune their knobs [54, 85, 105]. In contrast, BRAD optimizes an entire multi-engine data infrastructure instead of tuning individual services. BRAD can be seen as applying instance-optimization at the granularity of cloud database services instead of within a database engine [58].

Simplifying and optimizing the cloud. Like BRAD, recent research has explored ways to simplify and optimize the design and operation of cloud infrastructures. These thrusts include (i) high-level cloud programming abstractions [31, 67, 77, 94], (ii) infrastructure as code [18, 49], (iii) enhancing cross-cloud compatibility [30, 104], and (iv) improving resilience across services [66]. BRAD's key difference is that it focuses on simplifying cloud infrastructures containing multiple relational database systems while optimizing their use for cost under a performance constraint.

Single-system solutions. Another way to handle diverse data workloads is to use a single specialized (e.g., HTAP) database system designed for high performance across many workloads [40, 51, 56, 62, 98]. For some workloads (e.g., real-time analytics), such systems can be more efficient than BRAD because they are not internally constrained by engine boundaries. But these single-system solutions can be difficult to migrate to and they limit users to their specific feature set. In contrast, BRAD is designed to optimize existing multi-engine data infrastructures and (in concept) can include new systems to support specialized functionality (Section 5.2.3).

Polystores and federated databases. Prior work on polystores [3, 5, 37, 91, 107, 110, 118] and federated databases [25, 27, 28, 41, 52, 53, 75, 92, 97, 117] also aim to distribute query workloads across heterogeneous engines. Unlike BRAD, these systems focus on (i) optimizing queries within a given set of engines and hardware configuration, and (ii) bridging different data models [37]. In contrast, BRAD tackles the problem of selecting the best set of engines to include in the underlying infrastructure for the user's workload (among Aurora, Redshift, and Athena), while also jointly optimizing the workload assignment, engine provisioning, and data placement.

7 CONCLUSION

This paper presents *blueprints*, *blueprint planning*, and BRAD: a system that virtualizes a cloud data infrastructure and leverages blueprint planning to automatically manage its physical realization. The key takeaway is to cast infrastructure design as a *cost-based optimization problem*, which we refer to as *blueprint planning*. This approach allows us to systematically search for an optimized design for a given workload by leveraging learned models to predict the utility of candidate blueprints. We show that BRAD automatically achieves performance targets while saving 1.6–13× in cost compared to existing serverless autoscaling systems.

ACKNOWLEDGMENTS

This research was supported by Amazon, Google, and Intel as part of the MIT Data Systems and AI Lab (DSAIL) at MIT and NSF IIS 1900933. Geoffrey X. Yu was partially supported by an NSERC PGS D. This research was also sponsored by the United States Air Force Research Laboratory and the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. 2022. Proteus: Autonomous Adaptive Storage for Mixed Workloads. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. 700–714. https://doi.org/10.1145/3514221.3517834

[2] Ivo Adan and Jacques Resing. 2015. *Queueing Systems*. https://www.win.tue.nl/~iadan/queueing.pdf.

[3] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, Saravanan Thirumuruganathan, and Anis Troudi. 2018. RHEEM: Enabling Cross-Platform Data Processing: May the Big Data Be with You! *Proceedings of the VLDB Endowment* 11, 11 (2018), 1414–1427. https://doi.org/10.14778/3236187.3236195

[4] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE '12)*. 390–401. https://doi.org/10.1109/ICDE.2012.64

[5] Rana Alotaibi, Damian Bursztyn, Alin Deutsch, Ioana Manolescu, and Stamatis Zampetakis. 2019. Towards Scalable Hybrid Stores: Constraint-Based Rewriting to the Rescue. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1660–1677.

[6] Amazon Web Services. 2021. *Achieve up to 35% better price/performance with Amazon Aurora using new Graviton2 instances*. https://aws.amazon.com/about-aws/whats-new/2021/03/achieve-up-to-35-percent-better-price-performance-with-amazon-aurora-using-new-graviton2-instances/.

[7] Amazon Web Services. 2022. *AWS announces Amazon Aurora zero-ETL integration with Amazon Redshift* . https://aws.amazon.com/about-aws/whats-new/2022/11/amazon-aurora-zero-etl-integration-redshift/. Retrieved July 20, 2024.

[8] Amazon Web Services. 2023. *AWS announces Amazon Aurora I/O-Optimized*. https://aws.amazon.com/about-aws/whats-new/2023/05/amazon-aurora-i-o-optimized/. Retrieved July 20, 2024.

[9] Amazon Web Services. 2023. *How do I resize an Amazon Redshift cluster?* https://repost.aws/knowledge-center/resize-redshift-cluster. Retrieved July 20, 2024.

[10] Amazon Web Services. 2024. Amazon Athena. https://aws.amazon.com/athena/. Retrieved July 20, 2024.

[11] Amazon Web Services. 2024. *Amazon Athena Pricing*. https://aws.amazon.com/athena/pricing/. Retrieved July 20, 2024.

[12] Amazon Web Services. 2024. Amazon Aurora. https://aws.amazon.com/rds/aurora/. Retrieved July 20, 2024.

[13] Amazon Web Services. 2024. *Amazon Aurora Pricing*. https://aws.amazon.com/rds/aurora/pricing/. Retrieved July 20, 2024.

[14] Amazon Web Services. 2024. Amazon EC2. https://aws.amazon.com/ec2/. Retrieved July 20, 2024.

[15] Amazon Web Services. 2024. Amazon Redshift. https://aws.amazon.com/redshift/. Retrieved July 20, 2024.

[16] Amazon Web Services. 2024. *Amazon Redshift Pricing*. https://aws.amazon.com/redshift/pricing/. Retrieved July 20, 2024.

[17] Amazon Web Services. 2024. Amazon S3. https://aws.amazon.com/s3/. Retrieved July 20, 2024.

[18] Amazon Web Services. 2024. AWS CloudFormation. https://aws.amazon.com/pm/cloudformation/. Retrieved July 20, 2024.

[19] Amazon Web Services. 2024. AWS RDS Proxy. https://aws.amazon.com/rds/proxy/. Retrieved July 20, 2024.

[20] Amazon Web Services. 2024. Data Lakes and Analytics on AWS. https://aws.amazon.com/big-data/datalakes-and-analytics/. Retrieved July 20, 2024.

[21] Amazon Web Services. 2024. Purpose-Built Databases on AWS. https://aws.amazon.com/products/databases/. Retrieved July 20, 2024.

[22] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS '67 (Spring))*. 483–485. https://doi.org/10.1145/1465482.1465560

[23] Lyublena Antova, Derrick Bryant, Tuan Cao, Michael Duller, Mohamed A. Soliman, and Florian M. Waas. 2018. Rapid Adoption of Cloud Data Warehouse Technology Using Datometry Hyper-Q. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 825–839. https://doi.org/10.1145/3183713.3190652

[24] Pgbouncer Authors. 2024. Pgbouncer - Lightweight connection pooler for PostgreSQL. https://www.pgbouncer.org/. Retrieved July 20, 2024.

[25] Graham Bent, Patrick Dantressangle, David Vyvyan, Abbe Mowshowitz, and Valia Mitsou. 2008. A Dynamic Distributed Federated Database. In *Proceedings of the 2nd Annual Conference on International Technology Alliance (ACITA '08)*.

[26] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer.

[27] Yuri Breitbart, Hector Garcia-Molina, and Abraham Silberschatz. 1992. Overview of Multidatabase Transaction Management. *VLDB Journal* 1 (10 1992), 181–239. https://doi.org/10.1145/1925805.1925811

[28] Yuri Breitbart and Avi Silberschatz. 1988. Multidatabase Update Issues. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD '88)*. 135–142. https://doi.org/10.1145/50202.50217

[29] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. 2023. Tigger: A Database Proxy That Bounces with User-Bypass. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3335–3348. https://doi.org/10.14778/3611479.3611530

[30] Sarah Chasins, Alvin Cheung, Natacha Crooks, Ali Ghodsi, Ken Goldberg, Joseph E. Gonzalez, Joseph M. Hellerstein, Michael I. Jordan, Anthony D. Joseph, Michael W. Mahoney, Aditya Parameswaran, David Patterson, Raluca Ada Popa, Koushik Sen, Scott Shenker, Dawn Song, and Ion Stoica. 2022. The Sky Above The Clouds. arXiv:2205.07147 [cs.DC] https://arxiv.org/abs/2205.07147

[31] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. 2021. New Directions in Cloud Programming. arXiv:2101.01159 [cs.DC] https://arxiv.org/abs/2101.01159

[32] Yun Chi, Hyun Jin Moon, Hakan Hacigümüs, and Jun'ichi Tatemura. 2011. SLA-tree: A Framework for Efficiently Supporting SLA-based Decisions in Cloud Computing. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT '11)*. 129–140. https://doi.org/10.1145/1951365.1951383

[33] cppreference.com. 2024. C++ named requirements: Compare. https://en.cppreference.com/w/cpp/named_req/Compare. Retrieved July 20, 2024.

[34] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 215–226. https://doi.org/10.1145/2882903.2903741

[35] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 418–431. https://doi.org/10.1145/3448016.3457270

[36] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *Proceedings of the VLDB Endowment* 14, 2 (2020), 74–86. https://doi.org/10.14778/3425879.3425880

[37] Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, Magda Balazinska, Bill Howe, Jeremy Kepner, Sam Madden, David Maier, Tim Mattson, and Stan Zdonik. 2015. The BigDAWG Polystore System. *SIGMOD Rec.* 44, 2 (August 2015), 11–16. https://doi.org/10.1145/2814710.2814713

[38] Jennie Duggan, Olga Papaemmanouil, Ugur Çetintemel, and Eli Upfal. 2014. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 17th International Conference on Extending Database Technology (EDBT '14)*. 109–120. https://doi.org/10.5441/002/EDBT.2014.11

[39] Johannes Frnkranz and Eyke Hllermeier. 2010. *Preference Learning*. Springer-Verlag, Berlin, Heidelberg.

[40] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Engineering Bulletin* 35 (03 2012), 28–33.

[41] Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Amit P. Sheth. 1991. On Serializability of Multidatabase Transactions Through Forced Local Conflicts. In *Proceedings of the Seventh International Conference on Data Engineering (ICDE '91)*. 314–323.

[42] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Vol. 70. PMLR, 1263–1272. https://proceedings.mlr.press/v70/gilmer17a.html

[43] Google, Inc. 2024. BigQuery Omni. https://cloud.google.com/bigquery/docs/omni-introduction. Retrieved July 20, 2024.

[44] Google, Inc. 2024. Google Cloud Databases. https://cloud.google.com/products/databases. Retrieved July 20, 2024.

[45] Google, Inc. 2024. Google Compute Engine. https://cloud.google.com/compute. Retrieved July 20, 2024.

[46] Xingyu Gu, Adam Ronthal, Robert Thanaraj, and Julian Sun. 2024. Data and Analytics Cloud Adoption Survey Reveals Data Governance and Cost Challenges. Gartner Report. https://www.gartner.com/document/5106731.

[47] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *Proceedings of the VLDB Endowment* 15, 4 (2021), 752–765. https://doi.org/10.14778/3503585.3503586

[48] Mor Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.

[49] HashiCorp. 2024. Terraform. https://www.terraform.io. Retrieved July 20, 2024.

[50] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2361–2374. https://www.vldb.org/pvldb/vol15/p2361-hilprecht.pdf

[51] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[52] S.-Y. Hwang, E.-P. Lim, H.-R. Yang, S. Musukula, K. Mediratta, M. Ganesh, D. Clements, J. Stenoien, and J. Srivastava. 1994. The MYRIAD Federated Database Prototype. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*. https://doi.org/10.1145/191839.191986

[53] Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. 2002. Garlic: A New Flavor of Federated Query Processing for DB2. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. 524–532.

[54] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2953–2965. https://doi.org/10.14778/3551793.3551844

[55] Anastasios Karagiannis, Panos Vassiliadis, and Alkis Simitsis. 2013. Scheduling Strategies for Efficient ETL Execution. *Information Systems* 38, 6 (2013), 927–945.

[56] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[57] Ferdi Kossmann, Ziniu Wu, Eugenie Lai, Nesime Tatbul, Lei Cao, Tim Kraska, and Sam Madden. 2023. Extract-Transform-Load for Video Streams. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2302–2315. https://doi.org/10.14778/3598581.3598600

[58] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR '19)*. http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf

[59] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 489–504. https://doi.org/10.1145/3183713.3196909

[60] Tim Kraska, Tianyu Li, Samuel Madden, Markos Markakis, Amadou Ngom, Ziniu Wu, and Geoffrey X. Yu. 2023. Check Out the Big Brain on BRAD: Simplifying Cloud Data Processing with Learned Automated Data Meshes. *Proceedings of the VLDB Endowment* 16, 11 (8 2023), 3293–3301. https://doi.org/10.14778/3611479.3611526

[61] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2019. Learning to Optimize Join Queries With Deep Reinforcement Learning. arXiv:1808.03196 [cs.DB] https://arxiv.org/abs/1808.03196

[62] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Niloy Mukherjee, Atrayee Mullick, Sujatha Muthulingam, Vivekanandhan Raja, Marty Roth, Ekrem Soylemez, and Mohamed Zait. 2015. Oracle Database In-Memory: A Dual Format In-Memory Database. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering (ICDE '15)*. 1253–1258. https://doi.org/10.1109/ICDE.2015.7113373

[63] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[64] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems (NeurIPS '20)*.

[65] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1555–1566. https://doi.org/10.14778/2350229.2350269

[66] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. 2023. DARQ Matter Binds Everything: Performant and Composable Cloud Programming via Resilient Steps. *Proceedings of the ACM on Management of Data* 1, 2, Article 117 (2023), 27 pages. https://doi.org/10.1145/3589262

[67] Tianyu Li, Badrish Chandramouli, Sebastian Burckhardt, and Samuel Madden. 2024. Serverless State Management Systems. In *Proceedings of the Conference on Innovative Data Research (CIDR '24)*. https://www.cidrdb.org/cidr2024/papers/p16-li.pdf

[68] Wan Shen Lim, Matthew Butrovich, William Zhang, Andrew Crotty, Lin Ma, Peijing Xu, Johannes Gehrke, and Andrew Pavlo. 2023. Database Gyms. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '23)*.

[69] B. T. Lowerre. 1976. *The HARPY Speech Recognition System*. Ph.D. Dissertation. Carnegie Mellon University.

[70] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2022. Bao: Making Learned Query Optimization Practical. In *Proceedings of the International Conference on Management of Data (SIGMOD '22)*.

[71] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[72] Ryan Marcus and Olga Papaemmanouil. 2018. Deep Reinforcement Learning for Join Order Enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '18)*.

[73] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[74] Microsoft Corporation. 2024. Azure Compute. https://azure.microsoft.com/en-us/products/category/compute. Retrieved July 20, 2024.

[75] Microsoft Corporation. 2024. Microsoft Fabric Documentation. https://learn.microsoft.com/en-us/fabric/. Retrieved July 20, 2024.

[76] Elena Milkai, Yannis Chronis, Kevin P Gaffney, Zhihan Guo, Jignesh M Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. 1810–1824.

[77] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*. 561–577.

[78] Barzan Mozafari, Radu Alexandru Burcuta, Alan Cabrera, Andrei Constantin, Derek Francis, David Grömling, Alekh Jindal, Maciej Konkolowicz, Valentin Marian Spac, Yongjoo Park, Russell Razo Carranzo, Nicholas Richardson, Abhishek Roy, Aayushi Srivastava, Isha Tarte, Brian Westphal, and Chi Zhang. 2023. Making Data Clouds Smarter at Keebo: Automated Warehouse Optimization Using Data Learning. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 239–251. https://doi.org/10.1145/3555041.3589681

[79] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. 2013. Performance and Resource Modeling in Highly-Concurrent OLTP Workloads. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*.

[80] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 985–1000. https://doi.org/10.1145/3318464.3380579

[81] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[82] Jennifer Ortiz. 2019. *Performance-Based Service Level Agreements for Data Analytics in the Cloud*. Ph.D. Dissertation. University of Washington.

[83] Jennifer Ortiz, Brendan Lee, Magdalena Balazinska, Johannes Gehrke, and Joseph L Hellerstein. 2018. SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference ((USENIX ATC '18))*. 547–560.

[84] Jennifer Ortiz, Brendan Lee, Magdalena Balazinska, and Joseph L. Hellerstein. 2016. PerfEnforce: A Dynamic Scaling Engine for Analytics with Performance Guarantees. arXiv:1605.09753 [cs.DB]

[85] OtterTune, Inc. 2024. OtterTune | AI Powered Automatic PostgreSQL & MySQL Tuning. https://web.archive.org/web/20240605143522/https://ottertune.com/. Retrieved July 20, 2024.

[86] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '17)*. https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf

[87] Andrew Pavlo, Matthew Butrovich, Ananya Joshi, Lin Ma, Prashanth Menon, Dana Van Aken, Lisa Lee, and Ruslan Salakhutdinov. 2019. External vs. Internal: An Essay on Machine Learning Agents for Autonomous Database Management Systems. *IEEE Data Engineering Bulletin* (June 2019), 32–46. http://sites.computer.org/debull/A19june/p32.pdf

[88] Andrew Pavlo, Matthew Butrovich, Lin Ma, Wan Shen Lim, Prashanth Menon, Dana Van Aken, and William Zhang. 2021. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3211–3221. https://doi.org/10.14778/3476311.3476411

[89] Matthew Perron, Raul Castro Fernandez, David Dewitt, Michael Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. In *Proceedings of the ACM on Management of Data*, Vol. 1. Issue 4. https://doi.org/10.1145/3626720

[90] pgvector Authors. 2023. Open source vector similarity search for Postgres. https://github.com/pgvector/pgvector.

[91] Maksim Podkorytov and Michael Gubanov. 2019. Hybrid.Poly: A Consolidated Interactive Analytical Polystore System. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE '19)*. 1996–1999. https://doi.org/10.1109/ICDE.2019.00223

[92] Calton Pu. 1988. Superdatabases for Composition of Heterogeneous Databases. In *Proceedings of the Fourth International Conference on Data Engineering (ICDE '88)*. 548–555.

[93] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*. 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[94] Mingwei Samuel. 2021. *Hydroflow: A Model and Runtime for Distributed Systems Programming*. Master's thesis. University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-201.html

[95] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data (SIGMOD '23)*. 225–237. https://doi.org/10.1145/3555041.3589677

[96] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*. 23–34. https://doi.org/10.1145/582095.582099

[97] Amit P Sheth and James A Larson. 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys (CSUR)* 22, 3 (1990), 183–236.

[98] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. 731–742. https://doi.org/10.1145/2213836.2213946

[99] Snowflake, Inc. 2024. *ETL vs. ELT: Differences and Similarities*. https://www.snowflake.com/guides/etl-vs-elt. Retrieved July 20, 2024.

[100] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proceedings of the VLDB Endowment* 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296

[101] Transaction Processing Performance Council (TPC). 2024. *TPC-C*. https://www.tpc.org/tpcc/. Retrieved July 20, 2024.

[102] Transaction Processing Performance Council (TPC). 2024. *TPC-DS*. https://www.tpc.org/tpcds/default5.asp. Retrieved July 20, 2024.

[103] Transaction Processing Performance Council (TPC). 2024. *TPC-H*. https://www.tpc.org/tpch/default5.asp. Retrieved July 20, 2024.

[104] University of California, Berkeley. 2024. Sky Computing. https://sky.cs.berkeley.edu/. Retrieved July 20, 2024.

[105] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-Scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. 1009–1024. https://doi.org/10.1145/3035918.3064029

[106] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*. 363–378.

[107] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2018. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *Proceedings of the 2018 IEEE International Conference on Big Data (IEEE Big Data '18)*. 3364–3373.

[108] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[109] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *Proceedings of the International Conference on Management of Data (SIGMOD '21)*. 1879–1891. https://doi.org/10.1145/3448016.3457260

[110] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. 2017. The Myria Big Data Management and Analytics System and Cloud Services. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR '17)*.

[111] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE '13)*. 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899

[112] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data (SIGMOD '24)*. 280–294. https://doi.org/10.1145/3626246.3653391

[113] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: A New Cardinality Estimation Framework for Join Queries. *Proceedings of the ACM on Management of Data* 1, 1, Article 41 (2023), 27 pages. https://doi.org/10.1145/3588721

[114] Ziniu Wu, Pei Yu, Peilun Yang, Rong Zhu, Yuxing Han, Yaliang Li, Defu Lian, Kai Zeng, and Jingren Zhou. 2022. A Unified Transferable Model for ML-Enhanced DBMS. In *Proceedings of the 12th Conference on Innovative Data Systems Research (CIDR '22)*. https://www.cidrdb.org/cidr2022/papers/p6-wu.pdf

[115] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. *Proceedings of the VLDB Endowment* 16, 1 (2022), 99–112.

[116] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE '20)*. 1297–1308.

[117] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2022. Skeena: Efficient and Consistent Cross-Engine Transactions. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. 34–48. https://doi.org/10.1145/3514221.3526171

[118] Xiuwen Zheng, Subhasis Dasgupta, Arun Kumar, and Amarnath Gupta. 2022. AWESOME: Empowering Scalable Data Science on Social Media Data with an Optimized Tri-Store Data System. arXiv:2112.00833 [cs.DB]

[119] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1466–1479. https://doi.org/10.14778/3583140.3583160