Modular Monoliths: Agentic Analytical Database Architecture

Giuseppe Mazzotta
Firebolt
giuseppe@firebolt.io

SJ Saidi
Firebolt
jawad@firebolt.io

Mosha Pasumansky
Firebolt
moshap@firebolt.io

Benjamin Wagner
Firebolt
benjamin@firebolt.io

ABSTRACT

Coding agents are reshaping analytical databases on two fronts: the workloads they serve, and the way the databases themselves are built. Both changes push the architecture in the same direction: away from federated multi-tenant services and back toward a modular monolith. We describe how Firebolt is rebuilding its system around this principle, and the agent-driven engineering it unlocks.

KEYWORDS

cloud data warehouses, analytical databases, database architecture, coding agents, formal verification, Kubernetes operators

1 DATABASE DISRUPTION

Agents challenge the fundamental assumptions around database architecture. They will send virtually all queries, operate database systems, and implement new features across the system itself.

Modern cloud database warehouses are complex distributed systems with many moving parts [6, 14, 21], architected as multi-tenant managed services with many closed-source components. This was good architecture for the past, and it is also how we built Firebolt [19].

Running a full cloud data warehouse on a laptop today requires installing and managing many dependent services—metadata storage (e.g. FoundationDB), metadata services (e.g. snapshot compaction), garbage collection, and distributed caches [4, 9, 24].

These architectures will not stand the test of time. Agents will disrupt the database industry in two ways:

(1) *Agents **on** the database.* In the past, backends on top of database systems were static and human-written: tools in the modern data stack (e.g. dbt, Fivetran, Looker) leveraged by thousands of companies, or custom company-specific backends. The next era will see a Cambrian explosion of heterogeneous software systems and query patterns. As software costs drop to zero, custom app-specific backends will be common, and BI workloads will give way to agents running research and ad-hoc queries.

(2) *Agents **in** the database.* Defensibility of the database itself is eroding rapidly. As coding agents improve, they can also take on complex systems programming tasks; recent papers even synthesize workload-specific database systems from scratch [31]. The best database systems will be the ones easy for agents to modify, extend, and operate.

This disruption is an opportunity to completely reinvent modern data infrastructure, but it will fundamentally change database systems in ways that are incompatible with the architectures of modern cloud data warehouses.

SAO Workshop at CAIS'26, May 26, 2026, San Jose, California, USA
2026.

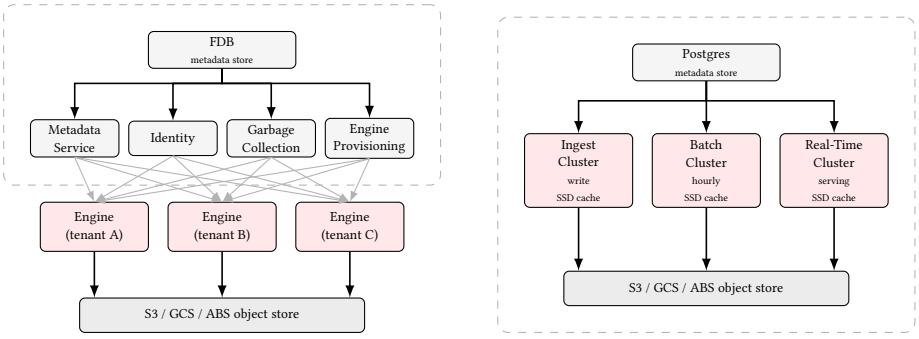
2 CHANGING WORKLOADS

As agents take over more of the workload sitting *on top of* analytical databases, the requirements those databases must satisfy are changing too. Talking to modern software and infrastructure engineering teams, we see three patterns emerge:

(1) **Rapid prototyping**: Historically, deploying a new database system required substantial time and effort to manage the underlying infrastructure, which directly slowed down iteration on new ideas. Most developers therefore experimented with only a narrow set of database systems (typically the ones they were already familiar with, given the learning curve of adopting a new one), or opted for a managed service entirely. In the agentic world, these barriers are substantially removed: (1) developers can iterate over more ideas, (2) try more database engines instead of defaulting to the familiar ones, and (3) realistically self-host the infrastructure instead of outsourcing it to a managed service.

(2) **Data privacy**: We see a growing number of companies running agentic workloads [23, 25] directly on top of data that contains Personally Identifiable Information (PII) [3]. At the same time, data privacy and sovereignty regulations [10, 26, 29] are continuing to tighten. This is in tension with the dominant deployment model: using a managed service typically requires ingesting customer data into the vendor's storage. For regulated industries, this incurs significant compliance overhead (e.g., listing the managed service as a data processor [10] and ensuring the required certifications such as SOC 2, ISO/IEC 27001, FedRAMP, or StateRAMP [2, 12, 27, 30]). This narrows the set of database systems that can realistically be considered.

(3) **Deployment flexibility**: Major cloud providers are expanding to new regions [1], and an increasingly heterogeneous set of new cloud providers is emerging to support new agentic workloads and to support data sovereignty requirements. Recent examples are AI-focused clouds such as Nebius [15] and CoreWeave [5], and sovereign European clouds such as STACKIT [22] and OVHcloud [18]. Beyond this, many organizations continue to run on-premise data centers. These increasingly look like modern cloud environments and support Kubernetes and object storage. The customer base is demanding more flexibility in where their data is stored, or processed, and the database system must be able to support this. For managed cloud data warehouses, expanding to new regions and cloud providers requires significant engineering effort and



(a) (b)

Dimension	Federated services	Modular monolith
Cost driver	Human eng. time	Agent iteration time
Op. surface	Many services	One binary + PG + obj. store
Multi-tenancy	Baked into components	Deployment-level config
Self-hosted	Per-region project	Single artifact

(c)

Figure 1: Firebolt's architectural transition: (a) federation of multi-tenant services on FDB; (b) modular monolith backed by Postgres and object storage; (c) trade-offs along the dimensions that drove the redesign.

time to market. Thus, using self-hosted databases is becoming an increasingly attractive option.

3 CHANGING DATABASE ECONOMICS

The other half of the disruption is internal: agents *in* the database, changing how the system itself gets built. That shift inverts the engineering economics that produced today's federated architectures in the first place.

For over a decade, cloud data warehouses lived inside a comfortable bubble. Building one was hard: only a handful of teams in the world had the systems expertise to ship a serious distributed query engine, and that scarcity translated directly into margin.

Scarcity shaped architecture. Past a certain team size, the cheapest way to ship a new capability—metadata, CDC tooling, governance, observability—was to spin up another standalone service, often in a different language than the core engine to broaden the talent pool. Firebolt's original architecture followed this playbook (C++ engine, Golang services). The modern data stack reflects this paradigm: a sprawling federation of products, each with its own runtime and failure mode.

Coding agents are popping this bubble. The constraint that justified federation—the high cost of human engineering time—is collapsing. Capabilities once kept out of the engine because they were too expensive to fold in can now be absorbed back into database internals. Margin pressure from new entrants and open source is accelerating the shift, and customers increasingly expect to run these systems anywhere rather than in a vendor's cloud. The economics behind architectural fragmentation are reversing (Figure 1c); architectures built on those old models should be questioned.

4 REINVENTING FIREBOLT

The original Firebolt cloud data warehouse was built for a multi-tenant, fully managed SaaS model. It was designed to host a large number of tenants, each running its own compute clusters, which we call Engines. Each Engine was a self-contained unit that could be started, stopped, scaled, and upgraded independently. Only the Engines were isolated per tenant. Every other major component of the system, such as metadata, garbage collection, identity, and engine provisioning, was shared across all tenants (Figure 1a). Each of these multitenant components was a standalone service, owned by different teams.

The decomposition gave each team a clear scope and room to specialize, but it imposed real costs on the system as a whole: failures in one component had a wide blast radius; team-level iteration slowed and release cycles stretched out, requiring extensive coordination across teams. Finally, customer-specific optimizations were hard to implement because many features may help some workloads, but hurt others.

The multitenant architecture also inflated component complexity. Our metadata layer, for example, was a set of microservices on FoundationDB (FDB), itself a complex distributed system requiring specialized operational expertise. This made sense for a multitenant metadata service that must scale extremely well, but it made the system hard to ship outside our own cloud: self-hosted deployments required rewriting several services, and each new region or cloud provider was a substantial engineering project.

Our prior architecture is a poor fit for the new world: the same friction that slowed our human teams (multiple services, ownership boundaries, longer release cycles) also limits what coding agents can accomplish on the system. With that in mind, we have started reinventing Firebolt around two goals: (1) the system should be easy for both humans and coding agents to extend and customize, and (2) it should serve as a self-contained backend for agentic applications across heterogeneous deployment environments, without sacrificing performance.

The new architecture is a deliberately simpler, modular monolith (Figure 1b). Subsystems that used to be standalone services—metadata, garbage collection, identity—are collapsed into the core database binary. Capabilities are opt-in: a deployment enables only what it needs, keeping the operational surface small. We built a Kubernetes operator that simplifies the deployment and automates compute lifecycle operations: provisioning, scaling, and upgrading clusters.

Crucially, *this architecture remains compatible with multi-tenant operation*, but on dramatically simpler terms. Because metadata lives in PostgreSQL rather than a bespoke microservice stack, a single Postgres deployment can back many Firebolt tenants at once, and innovative Postgres vendors like Neon [16] make this practical at

low cost via serverless, scale-to-zero services. PostgreSQL becomes the only truly shared service in managed multi-tenant deployments, reducing multi-tenancy to a deployment-level configuration rather than a property baked into every component.

5 A NEW DATABASE SOFTWARE LIFECYCLE

Architectural simplification is not just an aesthetic preference: it changes which engineering problems are tractable. Our goal is a software lifecycle where virtually all database and infrastructure code is agent generated. Turning the database into a modular monolith is a prerequisite for this.

Small, single-tenant components now do what used to require multitenant services. Garbage collection, metadata compaction, snapshot management, and distributed caching live inside the database itself, with cleaner interfaces and narrower scope because they no longer handle multitenancy. This enables strong agentic harnesses for autonomous coding in the core database layer.

The infrastructure layer wrapped around the database shrinks to a size that can be reasoned about formally. This is the precondition for an agent-driven software lifecycle, and it is structurally unavailable to architectures that did not start there [13].

This section gives an overview of how our streamlined architecture allows for faster, agent-driven systems engineering both for the database and infrastructure layer.

5.1 Metadata and Transaction Manager

The metadata and transaction manager is the system of record for every database, schema, table, view, role, and tablet in a Firebolt deployment. In the prior architecture this was a federation of Golang microservices on top of FoundationDB; in the new one it is a Rust component that runs in-process inside the database binary and persists to PostgreSQL. The component is a near-ideal target for agentic synthesis: it sits behind a narrow gRPC surface (transaction lifecycle, DDL, DML metadata, RBAC), it has well-defined consistency semantics (snapshot-isolated DDL, multi-statement transactions, MVCC visibility), and–most usefully–an externally-defined storage format we can borrow from.

DuckLake as a baseline. Rather than invent a new metadata schema, we built the metadata service on top of the DuckLake catalog format [8]. DuckLake is a recent, openly-specified lakehouse catalog whose entire metadata is stored in standard SQL tables. Firebolt extends the schema with a small set of tables and columns (multi-database scoping, server-side transaction tracking, additional objects), but the core schema is unchanged. This buys us interoperability with native DuckLake readers, and–more importantly for our purposes–an independent reference implementation we can use as a testing oracle.

Synthesis through a three-part harness. Our approach to having agents (re)write a system component is to wrap it in a harness strong enough that any behavioural regression surfaces immediately. The clean API boundary at the gRPC layer makes the harness easy to construct: the new component is a drop-in replacement for the old one, so the same workloads exercise both. For the metadata and transaction manager, the harness has three independent layers:

(1) **End-to-end test suite via the public API.** Firebolt's existing SQL test suite (thousands of `.test` files covering DDL, DML metadata, multi-statement transactions, and RBAC enforcement) runs unchanged against the new component. The synthesized metadata service answers the same gRPC calls as the old one, so the database binary cannot tell which implementation is behind the socket. A regression in observable behaviour fails a SQL test.

(2) **DuckDB's `ducklake` extension as a differential oracle.** For the DuckLake-compatible subset, we cross-check our catalog against an entirely independent implementation. A test is replayed twice: once through the synthesized metadata layer, and once through DuckDB's first-party `ducklake` extension against an isolated DuckDB catalog. After each test, the harness diffs the two catalogs row-by-row across every standard DuckLake table. Any divergence in object identity, column ordering, schema versioning, or snapshot lineage fails the build. This is differential testing against an oracle we did not author and do not control — exactly the property that makes such oracles trustworthy.

(3) **Catalog invariants as structural checks.** The DuckLake oracle covers only the standard subset of the schema; it tells us nothing about Firebolt's extensions or about the cross-table consistency of the catalog as a whole. We complement it with a structural validator that runs against the raw catalog and asserts invariants in four classes: *MVCC chain consistency*; *transaction lifecycle integrity*; *referential integrity across MVCC versions*; and *semantic uniqueness*. Each invariant is checked at every change and catches MVCC and commit-ordering bugs that the API-level tests miss.

What this enables. The three layers verify the *same* component from independent angles; a regression in any of them flags wrong synthesized code. The API-level suite proves the binary cannot tell new from old; the DuckLake oracle proves we have not silently diverged from the published format; the invariant checker proves the catalog stays internally consistent under arbitrary DDL interleavings. Together they form a complete, machine-checkable specification. Code synthesis then runs autonomously in a loop: the agent emits a candidate, the harness runs all three layers, every failure feeds the next iteration, and the loop terminates only when all layers pass. In total, this synthesis effort took less than a week. Human review shifts from inspecting code to inspecting the harness; once the harness is right, the synthesized code is right by construction. The same harness keeps the door open to future synthesis steps—swapping the storage backend, adding object types, rewriting the transaction protocol—each validated against the oracle and extensions of the harness.

5.2 Firebolt Kubernetes Operator

The Firebolt Kubernetes operator automates the lifecycle of Firebolt clusters. It can provision new clusters, rescale them, and perform online version upgrades.

The operator surface. The operator manages exactly two custom resources: the FireboltInstance for per-namespace shared state (e.g. PostgreSQL for metadata), and the FireboltEngine for stateful compute nodes with blue-green rollouts. The engine reconciler is a six-phase state machine—`stable`, `creating`, `switching`, `draining`, `cleaning`, `stopped`—with explicit rules for mid-flight spec changes

(abandon if `creating`, defer if `draining` or `cleaning`) and a hard dependency on a ready instance. That is the entire orchestration surface: no separate garbage-collection service, no metadata-snapshot operator, no cache-warmer sidecar—because no such services exist as separate processes.

Semi-formal verification in three phases. The small surface lets us apply a verification pipeline that would be impractical against federated microservices. This pipeline runs as a harness during agentic coding sessions, and in CI pipelines:

(1) **TLA+ specifications of the reconcilers.** Both control loops are modelled in PlusCal/TLA+. This is similar to lightweight industrial uses of formal modelling at AWS [17], MongoDB [20], and Datadog [7]. The engine spec (about 500 lines) covers all six phases, generation counters, every interleaving of user spec edits with reconcile steps, and crash points. The instance spec covers the sequential bring-up pipeline from Postgres to a ready metadata service. TLC checks safety invariants exhaustively. Example invariants are that no two generations serve traffic at once, the cluster service selector always matches the active generation, the quiesced terminal phase matches `spec.replicas`, and that every reachable state eventually reaches a terminal phase under fairness (liveness). Both specs check in seconds.

(2) **Stateful property tests against the live reconciler.** A Go property-based test drives the actual reconciliation loop with random sequences of spec changes, scale events, pod-readiness transitions, drain completions, and simulated crashes between resource writes and status updates. The invariants in the TLA+ spec are checked after every step. It runs on every PR and shrinks failures to minimal reproducers, giving us debuggable counter-examples in the same language the reconciler is written in.

(3) **TLA+ state cover against Go.** TLC dumps every reachable state of the model (1,386 states at `MaxGen=2, MaxSpec=3`). A code generator turns each state into a deterministic test that invokes `computeEngineReconcile` from that state and asserts the result lies in the reconciler's closure of the start state under the model. 892 states are exercised in well under a second. A CI guard regenerates the fixture on every push and fails the build if it diverges, so the model and the implementation cannot drift silently.

The crucial property is that all three layers verify the *same* invariants. A regression surfaces wherever the engineer happens to look first, and an architectural change to the reconciler cannot pass review without an accompanying update to the spec. None of this requires the engineer's day-to-day workflow to leave Go: TLA+ is a one-time investment per state machine, the property tests are vanilla `go test`, and the state-cover fixture is generated. The pipeline is complementary to recent academic work on full formal proofs for Kubernetes controllers [28] and on systematic mutation-based operator testing [11]: we trade exhaustive proof for a fast, in-tree feedback loop that an agent or engineer can iterate against on every commit.

What this enables. Verified, declarative infrastructure is the precondition for an agent-driven database software lifecycle. Once the operator can reliably bring a Firebolt instance from nothing to ready—and recover from any crash, spec edit, or pod failure—spinning one up becomes a primitive an agent can call. Agents can run in parallel, each claiming a fresh instance to validate a code change, replay a customer workload, shadow a production query, or A/B-test a query plan, then tearing it down when done. With the database collapsed into a single binary the operator can multiply at will, the experiment-per-agent model becomes the natural unit of work, and the same primitive backs internal development, customer-specific tuning, and on-prem deployment into regulated environments.

Why a redesign is required. Retrofitting this onto legacy cloud data warehouse architectures is, in our experience, substantially harder. Formally verifying the orchestration of dozens of services—each with its own state, deploy cadence, and multi-tenant isolation contract—is a fundamentally different problem, and multi-tenancy assumptions are typically baked too deep to dislodge cheaply. The fragmentation that produced margin and feature velocity in the 2010s is now the obstacle to the cost structure customers expect, to operations agents can keep running, and to deployment into customer environments (e.g. finance, defense, healthcare). We see this less as bolting agents onto the existing stack and more as redesigning the stack around them; vendors with deep federated investments may reasonably stage the transition. Small surface areas compound: an architecture small enough to verify is small enough for agents to safely modify, one safe to modify can be operated autonomously anywhere, and the one that runs autonomously anywhere is what customers want next.

6 CONCLUSION

Cloud data warehouses spent a decade fragmenting into complex federated services because human engineering time was scarce and deployment flexibility did not matter. Coding agents invert both assumptions: workloads shift toward heterogeneous, embedded, and on-premise deployments where sprawling control planes are a liability, and the development lifecycle now rewards architectures whose orchestration surface is small enough to specify, verify, and reproduce thousands of times a day. Both pressures point the same way: collapse federated services back into the database binary, keep the operational surface formally tractable, and let agents extend and operate the system. We do not see a clean preservation path through a fully federated architecture; the practical choice for new architectures is to start agent-native.

REFERENCES

[1] Amazon Web Services. 2026. Opening the AWS European Sovereign Cloud. AWS News Blog. https://aws.amazon.com/blogs/aws/opening-the-aws-european-sovereign-cloud/

[2] American Institute of Certified Public Accountants. 2017. Trust Services Criteria for Security, Availability, Processing Integrity, Confidentiality, and Privacy (SOC 2). AICPA TSP Section 100. https://www.aicpa-cima.com/resources/landing/system-and-organization-controls-soc-suite-of-services

[3] Shubhi Asthana, Ruchi Mahindru, Bing Zhang, and Jorge Sanz. 2025. Adaptive PII Mitigation Framework for Large Language Models. *arXiv preprint arXiv:2501.12465* (2025).

[4] ClickHouse. 2025. Building a Distributed Cache for S3. ClickHouse Blog. https://clickhouse.com/blog/building-a-distributed-cache-for-s3

[5] CoreWeave, Inc. 2025. CoreWeave: The AI Hyperscaler. Corporate website. https://www.coreweave.com/

[6] 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)*. ACM, 215–226. https://doi.org/10.1145/2882903.2903741

[7] Datadog. 2023. How Datadog Uses Formal Modeling, Lightweight Simulations, and Chaos Testing. Datadog Engineering Blog. https://www.datadoghq.com/blog/engineering/formal-modeling-and-simulation/

[8] DuckDB Labs. 2025. DuckLake: SQL as a Lakehouse Format. Manifesto. https://ducklake.select/manifesto/

[9] Pavan Edara and Mosha Pasumansky. 2021. Big Metadata: When Metadata is Big Data. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3083–3095. https://doi.org/10.14778/3476311.3476385

[10] European Parliament and Council of the European Union. 2016. Regulation (EU) 2016/679 (General Data Protection Regulation). Official Journal of the European Union, L 119, pp. 1-88. https://eur-lex.europa.eu/eli/reg/2016/679/oj

[11] Jiawei Tyler Gu et al. 2023. Acto: Automatic End-to-End Testing for Operation Correctness of Cloud System Management. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP '23)*.

[12] International Organization for Standardization. 2022. ISO/IEC 27001:2022: Information Security Management Systems Requirements. ISO/IEC Standard. https://www.iso.org/standard/27001

[13] Alp Keles, Jai Menon, Sesh Nalla, and Vyom Shah. 2026. Closing the Verification Loop: Observability-Driven Harnesses for Building with Agents. Datadog Blog. https://www.datadoghq.com/blog/ai/harness-first-agents/

[14] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3461–3472. https://doi.org/10.14778/3415478.3415568

[15] Nebius Group N.V. 2024. Nebius: An AI-centric Cloud Platform. Corporate website. https://nebius.com/

[16] Neon. 2024. Neon: Serverless Postgres. Corporate website. https://neon.tech/

[17] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (2015), 66–73. https://doi.org/10.1145/2699417

[18] OVHcloud. 2024. OVHcloud: Trusted Sovereign Cloud. Corporate website. https://ovhcloud.com/en/about-us/sovereign-cloud

[19] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a Query Engine From Spare Parts. In *Proceedings of the International Workshop on Cloud Data Management Systems (CDMS) at VLDB*. https://cdmsworkshop.github.io/2022/Slides/Fri_C2.5_MoshaPasumansky.pdf

[20] William Schultz, Ian Dardik, and Stavros Tripakis. 2021. eXtreme Modelling in Practice. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2129–2139. https://doi.org/10.14778/3476249.3476275

[21] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3731–3744. https://doi.org/10.14778/3685800.3685802

[22] Schwarz Group. 2022. Schwarz Group opens German Cloud STACKIT to Businesses and the Public Sector. Schwarz Group Press Release. https://gruppe.schwarz/en/press/archive/2022/schwarz-group-opens-german-cloud-stackit

[23] Alex Singla, Alexander Sukharevsky, Lareina Yee, Michael Chui, Bryce Hall, and Tara Balakrishnan. 2025. The State of AI in 2025: Agents, Innovation, and Transformation. McKinsey Global Survey on AI, QuantumBlack, AI by McKinsey. https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai

[24] Snowflake. 2024. How FoundationDB Powers Snowflake Metadata Forward. Snowflake Engineering Blog. https://www.snowflake.com/en/blog/how-foundationdb-powers-snowflake-metadata-forward/

[25] Stanford Institute for Human-Centered Artificial Intelligence. 2025. The AI Index 2025 Annual Report. Stanford HAI. https://hai.stanford.edu/ai-index/2025-ai-index-report

[26] State of California. 2018. California Consumer Privacy Act of 2018 (CCPA). Cal. Civ. Code Sec. 1798.100 et seq.. https://oag.ca.gov/privacy/ccpa

[27] StateRAMP. 2020. StateRAMP: Standardized Cybersecurity for State and Local Government. StateRAMP, Inc.. https://stateramp.org/

[28] Xudong Sun et al. 2024. Anvil: Verifying Liveness of Cluster Management Controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI '24)*.

[29] U.S. Congress. 1996. Health Insurance Portability and Accountability Act of 1996 (HIPAA). Public Law 104-191, 110 Stat. 1936. https://www.hhs.gov/hipaa/

[30] U.S. General Services Administration. 2011. Federal Risk and Authorization Management Program (FedRAMP). FedRAMP Program Management Office. https://www.fedramp.gov/

[31] Johannes Wehrstein et al. 2026. Bespoke OLAP: Synthesizing Workload-Specific One-size-fits-one Database Engines. *arXiv preprint arXiv:2603.02001* (2026).