ChampSoft
The Software Visionaries

Issue No 49 - September 1st 2026

NEWSLETTER

Volume 49



Data Engineering

Architecting the Modern Data Stack

For years, data engineering relied on rigid ETL (Extract, Transform, Load) pipelines. Data was extracted, transformed on fragile middleware servers, and finally loaded into rigid on-premise warehouses. The Modern Data Stack (MDS) inverts this into ELT (Extract, Load, Transform).

Page 01 - 03

Systems & Architecture

Designing Production-Ready Applications

Building software that works is only the starting point. As applications grow, the engineering challenge shifts from simply delivering features to designing systems that remain responsive, reliable, secure, and maintainable under real-world load

Page 04 - 07

champsoft.com — Blogs Live

Latest Blogs

Foward Deployed Engineer + Web Application Security + API - First Approach + AI in Modern Software Development

Architecting the Modern Data Stack

Snowflake, dbt, and Apache Airflow in production — a step-by-step blueprint for scalable, reliable, and testable data platforms.



For years, data engineering relied on rigid ETL (Extract, Transform, Load) pipelines. Data was extracted, transformed on fragile middleware servers, and finally loaded into rigid on-premise warehouses. The Modern Data Stack (MDS) inverts this into **ELT (Extract, Load, Transform)**. Today, raw data is ingested directly into a cloud warehouse, and transformations execute in-place using modular SQL.

This paradigm shift is anchored by three complementary technologies acting in unison to create a resilient, scalable, and automated data ecosystem.

01 — The Architectural Shift

From ETL to ELT

LAYER	TECHNOLOGY	CORE CAPABILITY
● Storage & Compute	Snowflake	Elastic data cloud with decoupled storage and compute.
● Transformation	dbt (Data Build Tool)	Version-controlled, modular SQL logic and testing.
● Orchestration	Apache Airflow	DAG-based scheduling, dependencies, and alerting.

To understand how these tools integrate, the following pages walk through the exact step-by-step lifecycle of a daily data pipeline in a production environment, then examine each of the three layers in turn.

02 — The Pipeline, Traced

Step-by-Step Execution Flow

To understand how these tools integrate, let's walk through the exact step-by-step lifecycle of a daily data pipeline in a production environment:

Step 01

Orchestration Trigger

Airflow initiates a daily Directed Acyclic Graph (DAG) based on a schedule or event trigger. It first instructs an ingestion tool (e.g., Fivetran or Airbyte) to extract new records from external operational databases.

Step 02

Raw Data Ingestion

The ingestion tool securely loads the raw, untransformed JSON or CSV data directly into an isolated RAW_STAGING database inside Snowflake.

Step 03

Staging Transformations

Airflow waits for a success signal from the ingestion tool, then triggers dbt. dbt executes its 'staging' models within Snowflake's compute engine to clean names, cast data types, and normalize formats without applying business logic.

Step 04

Business Logic & Marts

dbt executes the 'marts' models. It joins staged tables, applies business rules, and aggregates the metrics (e.g., merging user profiles with daily transaction histories).

Step 05

Data Contract Enforcement

Before final completion, dbt runs declarative schema tests (validating uniqueness, non-null constraints, and relationships) and custom SQL assertions. If a test fails, dbt halts the pipeline to prevent bad data from reaching production.

Step 06

Delivery & Analytics

The clean, modeled, and tested data now resides in an ANALYTICS schema within Snowflake, fully prepared for BI tools like Tableau, Looker, or AI/ML workloads.

03 — Storage & Compute

Snowflake: The Elastic Cloud Data Engine

Snowflake solves foundational infrastructure bottlenecks through its unique multi-cluster, shared-data architecture:

Decoupled Storage and Compute

By separating storage from compute clusters (Virtual Warehouses), Snowflake guarantees workload isolation. Heavy dbt transformation jobs run on dedicated warehouses without slowing down business executives querying dashboards. Virtual Warehouses scale instantly and auto-suspend when idle to minimize spend.

Zero-Copy Cloning

Data engineers can instantly create writable, isolated copies of production databases in seconds without duplicating physical storage. This enables flawless CI/CD workflows, as code changes can be tested against realistic data before merging.

Time Travel and Fail-safe

Snowflake natively preserves historical data snapshots (up to 90 days). Engineers can easily query past states or recover accidentally deleted tables instantly using the UNDROP command.

04 — Transformation

dbt: Software Engineering Rigor for Transformations

Operating strictly as a transformation compiler, dbt ensures SQL transformations are treated like mature software code:

Version-Controlled Modular Modeling

Business logic is broken into small, modular SQL files tracked in Git. This enforces code peer-reviews and prevents fragmented, undocumented stored procedures.

Jinja Templating & the ref() Function

Using Jinja templating via the {{ ref('model_name') }} function, dbt dynamically maps upstream dependencies to compile a complete execution DAG. Reusable macros eliminate repetitive code.

Declarative Testing

dbt enforces data reliability through built-in schema tests (unique, not_null, accepted_values). By defining tests in simple YAML files, teams guarantee data quality at the point of transformation.



05 — Orchestration

Apache Airflow: Enterprise Pipeline Orchestration

While Snowflake computes and dbt transforms, Apache Airflow serves as the master orchestrator, ensuring reliability across the ecosystem:

Topological DAG Execution

Airflow models complex workflows as Directed Acyclic Graphs (DAGs) in Python code, ensuring tasks run in absolute deterministic order. dbt models only execute when source data has successfully landed.

Operational Resilience & Alerting

Airflow manages transient network blips with automated retry policies and exponential backoff. In case of persistent failures, it automatically fires notifications to Slack or PagerDuty to alert on-call engineers.

Granular Orchestration with Cosmos

Modern architectures utilize the Astronomer Cosmos package to parse dbt projects and render each dbt model as a distinct task directly within the Airflow UI. This provides pinpoint observability and per-model retry capabilities.

Production Best Practices & Strategic ROI

Integrating this stack correctly delivers massive developer velocity and infrastructure cost savings. To maximize ROI, teams should implement the following production best practices:

01 Modular Scheduling

Run only the parts of the data stack that have changed, saving valuable time and cloud compute costs.

02 Environment Isolation

Utilize Snowflake's zero-copy cloning to maintain strict boundaries between Development and Production environments.



03 Continuous Integration (CI/CD)

Ensure every pull request triggers automated dbt compilations and tests before logic is ever merged into the main production branch.

"By replacing brittle, unmonitored scripts with Snowflake's elasticity, dbt's rigor, and Airflow's orchestration, data engineering teams can finally focus on shipping trustworthy data products that directly drive business value."



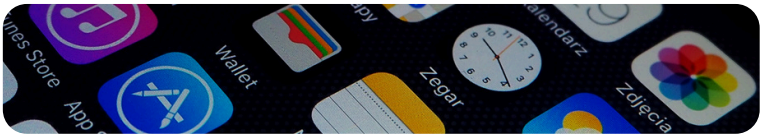
MANIKANTA GUNJI
Data Engineer
ChampSoft
The Software Visionaries

Designing Production-Ready Applications

APP

In the age of AI — practical architecture principles for building systems that scale, survive failure, and evolve

Building software that works is only the starting point. As applications grow, the engineering challenge shifts from simply delivering features to designing systems that remain responsive, reliable, secure, and maintainable under real-world load. The challenge becomes even more interesting when artificial intelligence is introduced. AI can add powerful capabilities, but it also introduces new constraints around latency, cost, throughput, data protection, and reliability.



01 Understand the Workload Before Choosing the Architecture

A million requests per second sounds impressive, but the number alone does not define the architecture. One million lightweight reads is very different from one million transactions that each require multiple database writes, external API calls, or AI inference.

Start by defining the traffic profile, peak load, latency targets, read/write ratio, consistency requirements, availability objectives, data sensitivity, and failure scenarios. Architecture should follow these requirements, not the popularity of a particular framework or platform.

Architect's question

What is our peak RPS, not just the average? What does each request actually do, and what happens if traffic suddenly becomes 10× higher?

THE SCENARIO

Consider a practical scenario: a platform expected to handle up to one million requests per second. The answer is not simply to buy more servers or immediately split everything into microservices. The real answer is to design the system so that work is distributed, expensive operations are isolated, failures are contained, and every layer can evolve independently.

02 Design the Request Path to Absorb Scale

A scalable application should distribute traffic before it reaches the most expensive components. Application services should remain as stateless as practical so that additional instances can be added when demand increases. Frequently requested data should be served from a distributed cache rather than repeatedly querying the database. The goal is to prevent the database or any single service from becoming the unavoidable bottleneck.

A typical request path

Client
↓
CDN / Edge
↓
Load Balancer
↓
Application Instances
↓
Cache
↓
Data Services

03 Keep the Critical Path Small

Users should not have to wait for work that does not need to happen before a response can be returned. Imagine an order system that must save an order, send notifications, update analytics, generate a report, and create an AI summary. Performing all of those tasks synchronously makes every dependency part of the user's latency.

A better design completes the critical transaction first and moves non-critical work to queues and background workers. This improves responsiveness and allows the platform to absorb traffic bursts without overwhelming downstream systems.

Architect's question

Does this operation really need to happen before we respond, or can it safely happen asynchronously?

04 Choose Monoliths and Micro services for a Reason

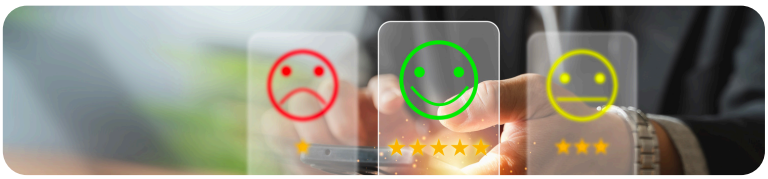
A production system does not have to be microservices-based to scale. A well-structured modular monolith can be horizontally scaled and can often be simpler to test, deploy, and operate.

Microservices become valuable when independent deployment, ownership, scaling, or technology choices justify the additional distributed-system complexity. A useful progression can be: modular monolith → horizontally scaled application → extract genuine bottlenecks → selectively introduce services.

The important design principle is simple: create strong boundaries first; distribute them when there is a real reason to do so.

Architect's question

Does this capability have a clear boundary? Does it really need independent scaling or deployment, and what happens if it goes down?



05 Treat AI as a Capability, Not the Architecture

Traditional applications often follow a path such as API → Business Logic → Database. An AI-enabled application adds another concern: API → Business Logic → AI Orchestration → Context Retrieval → Model.

The AI orchestration layer can manage prompt construction, retrieval, model selection, guardrails, response validation, timeouts, retries, fallbacks, and model versions. Keeping these responsibilities behind a clear boundary prevents every business component from becoming tightly coupled to one AI provider or model.

This also makes the architecture easier to evolve. A model can be replaced, upgraded, or routed differently without rewriting the core business logic.

Architect's question

Does this problem require AI? What data does the model need, and can we replace the model without rewriting business logic?

06 Do Not Send Every Problem to the Largest Model

AI introduces a different kind of scalability problem. Model inference can be expensive and comparatively slow, so the best architecture does not treat every request equally.

Simple requests may use a smaller model. Repeated requests can use semantic or conventional caching. Complex reasoning can be routed to a more capable model. Long-running analysis can move to an asynchronous workflow. Streaming can improve the perceived responsiveness of interactive experiences.

This turns AI architecture into a balancing act between quality, latency, throughput, and cost.

Architect's question

If we receive one million requests, how many need AI? Could rules, caching, a smaller model, or asynchronous processing handle the rest?

07 Design for Failure Before Failure Happens

At high scale, failure is not an unusual event. Databases become slow, networks drop packets, queues grow, third-party services rate-limit clients, and AI providers occasionally become unavailable.

Production systems should therefore use timeouts, bounded retries with backoff, circuit breakers, rate limiting, idempotency, health checks, dead-letter queues, and graceful degradation.

For AI systems, graceful degradation is especially important. If an AI capability becomes unavailable, the core application should continue operating where possible instead of allowing an optional intelligence feature to become a single point of failure.

Architect's question

If a critical dependency disappeared for 30 minutes, could the rest of the application still provide useful functionality?

08 Make Observability Part of the Design

At scale, optimization based on assumptions is dangerous. Engineers need evidence. Useful signals include requests per second, error rates, p95/p99 latency, cache hit ratios, database latency, queue depth, resource utilization, and dependency failures.

AI systems add another layer of visibility: model latency, token consumption, model failure rates, prompt/model versions, and cost per request. Distributed tracing becomes increasingly valuable when a single request crosses several services.

Architect's question

If the system becomes slow at 2 AM, can an engineer quickly identify which component is responsible?

09 Build Security Into the Architecture

Security should not be a final checklist item. Authentication, authorization, encryption, secret management, input validation, API protection, audit logging, and data access controls should be considered while the system is being designed.

AI adds further considerations, including protecting sensitive information sent to models, controlling access to AI capabilities, validating generated output, and defending against malicious or manipulated prompts. For regulated or sensitive workloads, these concerns must influence the architecture from the beginning.

Architect's question

What is the most sensitive data, where does it travel, and what prevents an AI capability from accessing data it should never see?

10 Design for Evolution, Not Prediction

No team can predict every future requirement. A better strategy is to create an architecture that can change without forcing a complete rewrite.

A system might begin as a modular monolith, scale horizontally as usage grows, introduce caching and asynchronous processing, and later extract only the components that genuinely need independent scaling. AI capabilities can follow the same path: start with a well-defined boundary, measure actual usage, and introduce specialized infrastructure only when the workload justifies it.

Architect's question

If traffic, data, or business requirements grow by 10×, what will fail first, and can we change that part without rebuilding everything?



The Practical Takeaway

The goal of scalable architecture is not to use the most technologies. It is to make thoughtful decisions about where work happens, how traffic is distributed, what happens when something fails, and how the system can change over time.

Whether the application is a monolith, a collection of microservices, or an AI-enabled platform, the fundamentals remain consistent: establish clear boundaries, keep the critical path lean, scale horizontally, cache intelligently, process expensive work asynchronously, observe the system continuously, and design every important dependency with failure in mind.

The Architect's Final Checklist

Before approving a design, ask: Can it scale? Can it survive failure? Can we observe it? Can we secure it? Can it evolve? And finally: are we solving today's problem with tomorrow's complexity?

Good architecture is not about having the most boxes on a diagram or the most technologies in the stack. It is about making sensible trade-offs for the problem we actually have, while leaving enough room for the system to grow.

AI does not replace these principles; it makes them more important. The strongest production applications will combine sound software architecture with AI capabilities that are deliberately designed, observable, secure, and resilient.

"AI does not replace good software engineering. It raises the bar for it."



SHAMMI SEPALA
Senior Software Engineer
ChampSoft
The Software Visionaries



CELEBRATING 13 Years OF EXCELLENCE

INNOVATING TOGETHER, GROWING TOGETHER

We celebrate more than just the number of years, We celebrate the people, passion, and purpose that have shaped our journey.

OUR PEOPLE
The Heart of ChampSoft

OUR PASSION
Driving Innovation and Impact

OUR PURPOSE
Building Solutions that Matters






















13
Years




13
Years


13
Years


13
Years


13
Years

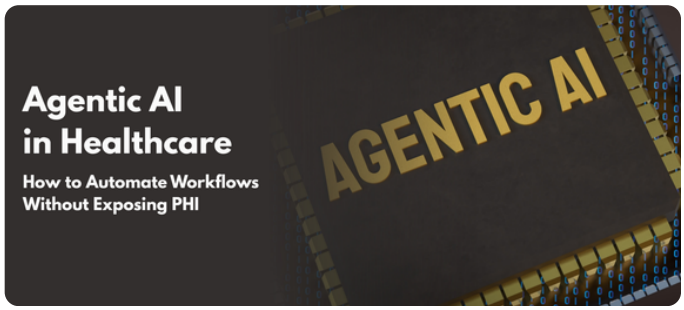
CATCH OUR LATEST BLOGS

Read the Latest on - champsoft Blogs



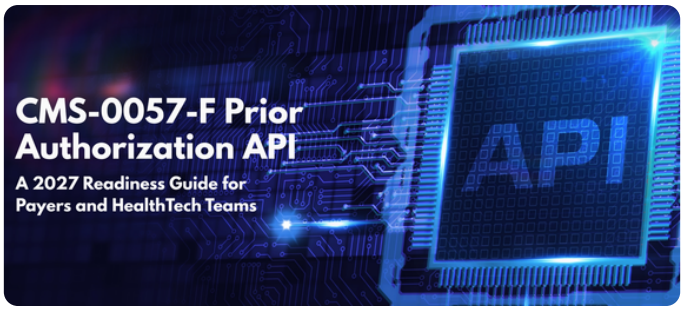
AI Software Development Cost in 2026: Pricing for AI Agents, LLM Apps, Machine Learning and Enterprise Platforms

AI software development in 2026 typically costs $15,000–$50,000 for a simple AI agent or chatbot, $80,000–$250,000 for a production LLM application or custom machine learning system,...



Agentic AI in Healthcare: How to Automate Workflows Without Exposing PHI

Agentic AI in healthcare uses goal-driven software agents to plan tasks, access approved systems, and take actions across clinical or administrative workflows. To protect PHI,...



CMS-0057-F Prior Authorization API: A 2027 Readiness Guide for Payers and HealthTech Teams

CMS-0057-F requires affected payers to implement or enhance FHIR-based interoperability APIs, including a Prior Authorization API, generally beginning January 1, 2027.



AI Custom Software Development: A Comprehensive Guide for Businesses

AI custom software development is the process of building AI-powered applications from machine learning models to AI agents and LLM integrations designed specifically around a business's workflows



How Hospital Risk Management Software Transforms Patient Safety

Hospital risk management software can improve patient safety by replacing fragmented or manual incident-reporting processes with centralized tracking, risk analysis, workflow automation,



MCP vs A2A: How Enterprise AI Agents Connect to Tools, Data, and Each Other in 2026

MCP and A2A solve different integration problems. Model Context Protocol standardizes how an AI application or agent accesses tools, data, and reusable capabilities. Agent2Agent