Building Enterprise-Ready Data Platforms: Core Handbook

This core handbook provides a high-level overview of building enterprise-ready data platforms, focusing on the fundamental principles, architectural choices, and the progressive path to a robust data ecosystem. It's designed for busy engineers and executives who need a concise understanding of the platform's purpose, structure, and key decision points without wading through exhaustive technical details.

1. Purpose and Introduction

This guide is meticulously crafted for experienced Data Engineers and Senior Software Engineers tasked with modernizing enterprise data ingestion stages. It provides a practical, hands-on approach to building a robust local development environment that mirrors a scalable, production-grade data platform. The focus is on developing Python-based ETL pipelines for disparate data sources, including simple financial and insurance data, emphasizing modern architectural patterns and best practices.

1.1. Why a Robust Local Environment?

A robust local development environment is paramount for building enterprise-ready data platforms. It enables rapid iteration, extensive testing, and critical skill development without incurring cloud costs or dependencies. This approach significantly de-risks subsequent cloud deployments, accelerates development cycles, and allows engineers to experiment with complex distributed systems in a controlled, isolated setting.

From a business perspective, this translates directly into:

- **Faster Time-to-Value:** Rapid prototyping and local testing accelerate the delivery of new data products and features.
- **Reduced Cloud Spend:** Significant cost savings during the development and testing phases by minimizing reliance on expensive cloud resources.
- **Enhanced Audit Trails & Compliance Readiness:** A controlled environment facilitates the implementation and testing of governance features from day one, bolstering compliance efforts.

For more in-depth coverage of testing strategies, refer to the **Testing & Observability Patterns Deep-Dive Addendum**. For details on automating deployments, see the **IaC & CI/CD Recipes Deep-Dive Addendum**.

1.2. The Progressive Complexity Path

To avoid the "all-or-nothing" overwhelm often associated with complex data platforms, this guide introduces a "Progressive Complexity" path. Engineers can ramp up feature-by-feature, mastering core components before integrating more advanced elements. This structured approach:

- **Reduces Cognitive Load:** By introducing components incrementally, engineers can focus on understanding one set of interactions at a time.
- **Accelerates Learning:** Hands-on experience with foundational elements provides a solid base for more complex systems.
- **Facilitates Skill Development:** Engineers can gradually build expertise across the entire data platform stack.
- **Enables Flexible Development:** Teams can choose the appropriate track based on their current project needs and scale requirements.

The tracks are designed as follows:

- **Starter Track:** Focuses on a minimal, single-machine setup for foundational data ingestion and storage. Perfect for initial prototyping and simple use cases.
- **Intermediate Track:** Introduces real-time streaming capabilities with Apache Kafka and distributed processing with Apache Spark. Ideal for addressing real-time data needs and scaling transformations.
- **Advanced Track:** Integrates the full suite of tools for comprehensive orchestration, lineage, observability, and metadata management, culminating in a production-grade local environment. This is for building highly robust and governable data platforms.

For a detailed setup guide for each of these tracks, refer to the **Progressive Path Setup Guide Deep-Dive Addendum**.

1.3. Embracing the Modern Data Engineer Role

Data ingestion represents the critical first step in any data-driven organization, transforming raw, disparate data into actionable insights. This document explores both ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) methodologies, highlighting their application in different scenarios. By replicating a production-like setting locally—simulating AWS Lambdas with SAM, alongside distributed components like Apache Spark and Apache Kafka—this guide empowers practitioners to ensure smooth transitions and reduced friction when deploying solutions to the cloud.

Furthermore, the comprehensive scope of this guide, encompassing Python, Docker, distributed systems, observability, data lineage, and machine learning (ML) elements, reflects a significant evolution in the role of a data engineer. This expanded scope moves beyond traditional data movement tasks to encompass broader responsibilities in system design, operational excellence, and leveraging advanced analytics. The task of rewriting a data ingestion stage for a large company, involving several disparate data sources, inherently implies a need for handling complexity and scale. This guide aims to empower the practitioner to demonstrate this expanded skill set, beneficial not just for platform builders but also for data scientists and analysts who will consume data from it. For a deeper dive into these areas, please refer to the respective deep-dive addendums: **IaC & CI/CD Recipes**, **Testing & Observability Patterns**, **DR & Runbooks**, and **Cloud Migration + Terraform Snippets**.

2. Executive Summary: Platform Pitch & ROI

This section provides a high-level overview of the proposed data platform, designed for internal pitching to stakeholders, highlighting key benefits and estimated returns on investment.

Building an Enterprise-Ready Data Platform: Strategic Imperative

- **Problem:** Our current data infrastructure struggles with diverse, high-volume data ingestion, real-time analytics, and comprehensive data governance, leading to slow insights, high operational costs, and compliance risks.
- **Solution:** Implement a modern, scalable, and observable data platform leveraging open-source technologies for a robust local development environment, seamlessly transitioning to a cost-efficient cloud-native architecture.

Key Business Benefits & ROI:

Benefit Area	**Current State Challenge**	**Platform Impact**	**Estimated ROI / Metrics**
Faster Time-to-Value	Manual setup, slow iteration cycles	~30% faster feature delivery due to robust local dev & CI/CD. Reduces developer onboarding from weeks to days.	Reduce feature delivery time by 3-4 weeks/quarter. Accelerate new data product launches.
Reduced Cloud Spend	Inefficient resource utilization in dev/test	~20-40% reduction in dev/test cloud costs by shifting workloads locally. Optimized cloud resource scaling.	Anticipated $50K-$150K annual savings in non-production cloud infrastructure.
Improved Data Quality	Inconsistent data, manual validation	Automated schema enforcement, contract testing, and data quality checks (e.g., Great Expectations).	Decrease data-related incidents by 50%, ensuring reliable insights and reporting.
Enhanced Compliance & Audit	Scattered data, unclear lineage, manual audits	Centralized metadata catalog (OpenMetadata), automated lineage (Spline), and robust access controls.	Streamline audit preparation by 70%, reducing compliance burden and risk penalties.
Operational Efficiency	Reactive issue resolution, alert fatigue	Proactive monitoring (Grafana), clear SLIs/SLOs, and structured incident response (DR Playbook).	Reduce Mean Time To Resolution (MTTR) by 40% for data-related incidents. Lower operational overhead.
Scalability & Future-Proofing	Monolithic systems, limited real-time	Modular, distributed architecture (Kafka,	Support 5x data growth over 3 years

	capabilities	Spark, Delta Lake) built for petabyte scale and real-time processing.	without major re-architecture. Enable new real-time use cases (e.g., fraud detection).

*For more details on CI/CD benefits, refer to the **IaC & CI/CD Recipes Deep-Dive Addendum**. For insights into data quality, operational efficiency, and observability, see the **Testing & Observability Patterns Deep-Dive Addendum**. For information on disaster recovery and MTTR, consult the **DR & Runbooks Deep-Dive Addendum**.*

Project Milestones (Conceptual):

- **Q3 202X:** Establish core local dev environment (Starter & Intermediate Tracks).
- **Q4 202X:** Implement full Advanced Track locally, complete initial CI/CD pipelines.
- **Q1 202Y:** Pilot AWS migration for a critical data ingestion pipeline.
- **Q2 202Y:** Full production rollout on AWS.

*For detailed guidance on cloud migration, see the **Cloud Migration + Terraform Snippets Deep-Dive Addendum**.*

Ask: Secure resources for dedicated engineering focus to implement this strategic platform modernization, unlocking significant business value and long-term capabilities.

3. The Progressive Path to an Enterprise Data Platform

This section details the step-by-step approach to building the local data platform, starting simple and progressively adding complexity. Each track builds upon the previous one. For a detailed setup guide for each of these tracks, including docker-compose.yml configurations, refer to the **Progressive Path Setup Guide Deep-Dive Addendum**.

3.1. Starter Track: Minimal Single-Machine Setup

The starter track provides the bare essentials for data ingestion and structured storage, ideal for rapid prototyping and understanding fundamental data flow. This minimal setup requires low computational resources and serves as an excellent entry point for engineers new to the platform.

Components:

- **FastAPI:** A lightweight, high-performance web framework for building data ingestion APIs.
- **PostgreSQL:** A robust relational database for structured data and API-specific metadata.
- **MinIO (as File-based Delta Lake):** An S3-compatible object storage server, simulating a data lake where immutable Delta Lake files reside.

Key Learnings:

- **API Design:** How to create secure and well-documented endpoints for data reception using FastAPI.
- **Database Interaction:** Storing and retrieving structured data efficiently with

PostgreSQL.
- **Object Storage Basics:** Understanding the S3-compatible interface for local data lake operations with MinIO.
- **Containerization Fundamentals:** Running individual services in isolated Docker containers.
- **Direct Storage Patterns:** Simple ETL/ELT patterns where data is written directly to a database or object store.

3.2. Intermediate Track: Adding Streaming Capabilities

This track expands the platform to handle real-time data streams and distributed transformations. It introduces two powerful, industry-standard components that form the backbone of many modern data architectures.

Components (in addition to Starter):

- **Apache Kafka:** A distributed streaming platform for high-throughput, fault-tolerant real-time data ingestion and event streaming. It decouples producers from consumers.
- **Apache Spark:** A powerful, distributed processing engine for large-scale data transformations, supporting both batch and streaming workloads. It will consume data from Kafka and write to Delta Lake.

Key Learnings:

- **Asynchronous Ingestion:** Decoupling producers and consumers using a message broker like Kafka for resilience and scalability.
- **Distributed Stream Processing:** Consuming from Kafka and writing to Delta Lake with Spark Structured Streaming, enabling near real-time data pipelines.
- **Data Lakehouse Concepts:** Implementing ACID transactions, schema enforcement, and time travel capabilities with Delta Lake on object storage.
- **Scaling Data Pipelines:** Understanding the basics of distributed systems and how Spark partitions and processes data across workers.

3.3. Advanced Track: The Full Production-Ready Stack

The advanced track integrates robust solutions for orchestration, observability, lineage, and metadata management, simulating a comprehensive enterprise-grade platform. This track represents the complete vision for the local development environment, providing all the tools necessary for building, monitoring, and governing complex data ecosystems.

Components (in addition to Intermediate):

- **Apache Airflow:** Workflow orchestrator for scheduling and managing complex data pipelines and their dependencies.
- **OpenTelemetry & Grafana Alloy:** Standardized telemetry collection and forwarding, enabling comprehensive monitoring.
- **Grafana:** Interactive data visualization and monitoring dashboards for operational insights.
- **Spline:** Automated data lineage tracking specifically for Spark jobs, providing visibility into data transformations.
- **OpenMetadata:** Comprehensive metadata management and data cataloging,

consolidating information from various sources.
- **MongoDB:** A flexible NoSQL document database, suitable for semi-structured data or specific application use cases requiring schema flexibility.
- **cAdvisor:** Container resource usage and performance analysis agent, providing metrics for Grafana.

Key Learnings:

- **Orchestration Mastery:** Managing complex workflows and dependencies with Airflow, including scheduling Spark jobs and metadata ingestion tasks.
- **End-to-End Observability:** Gaining deep insights into system health, performance, and bottlenecks using OpenTelemetry, Grafana Alloy, Grafana, and cAdvisor. For detailed patterns, refer to the **Testing & Observability Patterns Deep-Dive Addendum**.
- **Data Lineage & Governance:** Tracking data transformations with Spline and providing a unified data catalog for discovery, understanding, and compliance with OpenMetadata.
- **Comprehensive Data Management:** Integrating diverse data stores (relational, NoSQL, object storage) and tools for a holistic, enterprise-ready data platform.

4. Foundational Architecture & Core Technologies

This section provides a concise, high-level overview of the platform's architecture and the core technologies integrated into the local data platform, outlining each component's primary function and contribution to the overall scalable system. Docker Compose is the pivotal tool for managing the interdependencies and orchestration of this complex local data stack, simplifying the simulation of distributed systems. For the full docker-compose.yml and project structure, refer to the **IaC & CI/CD Recipes Deep-Dive Addendum**.

The proposed architecture transforms data pipelines into a scalable, distributed system, adopting the "data lakehouse" paradigm. By leveraging Delta Lake as the primary storage layer, the architecture creates a unified solution for both raw and curated data, simplifying the system, reducing redundancy, minimizing data movement, and ensuring data consistency. The introduction of Apache Kafka and Spark Structured Streaming addresses the need for real-time analytics, critical for immediate analysis in security, financial, or insurance scenarios. The decoupling of ingestion from storage via Kafka significantly improves the resilience and availability of the ingestion layer by buffering events and preventing backpressure.

4.1. Architectural Overview

The platform is logically divided into several layers:

- **Ingestion Layer:** The entry point for all raw data, handling external data sources and publishing to a streaming buffer.
- **Processing Layer:** Where data is transformed, cleansed, validated, and modeled using distributed computing.
- **Storage Layer (Data Lakehouse):** The unified, reliable repository for all data states (raw, curated), providing ACID properties and flexible schema management.

- **Analytical Layer:** Facilitates querying, reporting, and advanced analytics, including machine learning model training and inference.
- **Orchestration & Governance Layer:** Manages workflow scheduling, ensures data quality, provides end-to-end observability, and offers a centralized data catalog with lineage capabilities.

For a detailed mapping of local components to AWS cloud services, refer to the **Cloud Migration + Terraform Snippets Deep-Dive Addendum**.

Here is a PlantUML diagram illustrating the architectural overview:

```
@startuml
!theme toy
skinparam componentStyle uml2

' Define Actors/External Systems
actor "Disparate Data Sources\n(e.g., Financial, Insurance Systems)" as data_sources

' Define Layers/Zones
rectangle "Ingestion Layer" {
  component "FastAPI Ingestor" as fastapi_ingestor
  queue "Apache Kafka\n(Raw Data Topic)" as kafka_topic
}

rectangle "Processing Layer" {
  component "Apache Spark Cluster" as spark_cluster
  rectangle "Spark Structured Streaming\n(Raw Data Consumer)" as spark_raw_consumer
  rectangle "PySpark Transformation Job\n(ELT/Batch)" as spark_transform
  spark_cluster -- spark_raw_consumer
  spark_cluster -- spark_transform
}

rectangle "Storage Layer (Data Lakehouse)" {
  database "MinIO (S3 Compatible)\n(Delta Lake Raw Zone)" as minio_raw
  database "MinIO (S3 Compatible)\n(Delta Lake Curated Zone)" as minio_curated
  database "PostgreSQL\n(Structured Data/Metadata)" as postgres_db
  database "MongoDB\n(Semi-Structured Data)" as mongodb_db
  minio_raw <--> minio_curated : "Delta Lake"
}

rectangle "Orchestration & Governance Layer" {
  cloud "Apache Airflow" as airflow
  component "OpenTelemetry" as opentelemetry
  component "Grafana Alloy\n(OTLP Collector)" as grafana_alloy
  database "OpenMetadata\n(Data Catalog)" as openmetadata
  component "Spline\n(Spark Lineage)" as spline
```

```
    component "Grafana\n(Monitoring & Visualization)" as grafana
    component "cAdvisor\n(Container Metrics)" as cadvisor
}

rectangle "Analytical Layer" {
    component "Spark SQL / MLlib Analytics" as spark_analytics
}

' Data Flow
data_sources --> fastapi_ingestor : "Send Data (HTTP/S)"
fastapi_ingestor --> kafka_topic : "Publish Data (JSON/Protobuf)"
kafka_topic --> spark_raw_consumer : "Consume Stream"
spark_raw_consumer --> minio_raw : "Write to Raw Zone"
minio_raw --> spark_transform : "Read Raw Data"
spark_transform --> minio_curated : "Write Curated Data (MERGE)"
minio_curated --> spark_analytics : "Query for Analytics"
postgres_db <--> spark_transform : "Dim Data / Metadata"
mongodb_db <--> spark_transform : "Semi-Structured Data"
spark_analytics --> data_sources : "Insights/Reports"

' Observability Flow
opentelemetry --> grafana_alloy : "Telemetry Data (Traces, Metrics, Logs)"
fastapi_ingestor .. opentelemetry : "Instrumented"
spark_cluster .. opentelemetry : "Instrumented"
airflow .. opentelemetry : "Instrumented"
cadvisor --> grafana_alloy : "Container Metrics"
grafana_alloy --> grafana : "Forward to Grafana"
grafana_alloy --> openmetadata : "Forward Metadata/Telemetry"
spark_cluster --> spline : "Capture Lineage"
spline --> openmetadata : "Send Lineage Metadata"
airflow --> spark_cluster : "Orchestrate Jobs"
airflow --> openmetadata : "Orchestrate Metadata Ingestion"
openmetadata <--> grafana : "Share Metadata/Context"
@enduml
```

4.2. Core Technology Deep Dive

The following summarizes the key technologies and their roles within this platform, directly correlating to the services defined in the docker-compose.yml. For the complete docker-compose.yml and detailed setup instructions, refer to the **IaC & CI/CD Recipes Deep-Dive Addendum**. For cloud-native replacements and their Terraform snippets, see the **Cloud Migration + Terraform Snippets Deep-Dive Addendum**.

- **Apache Airflow:** Workflow orchestrator for scheduling, monitoring, and managing complex data pipelines and dependencies, including Spark jobs. Provides a robust framework for defining complex data pipelines as Directed Acyclic Graphs (DAGs).

- **Apache Kafka:** A distributed streaming platform designed for building real-time data pipelines and streaming applications. It serves as a durable buffer for raw event streams, decoupling ingestion from downstream processing.
- **Apache Spark:** A powerful, distributed processing engine for large-scale data transformations (ELT), supporting both batch and streaming workloads with PySpark. It reads from Kafka and Delta Lake.
- **AWS SAM CLI:** (Serverless Application Model Command Line Interface) Enables local development and testing of AWS Lambda functions, simulating the serverless environment.
- **cAdvisor:** (Container Advisor) A running daemon that collects, aggregates, processes, and exports information about running containers, providing performance metrics to Grafana.
- **Delta Lake:** An open-source storage layer that brings ACID transactions, schema enforcement, and time travel capabilities to data lakes, unifying batch and streaming data processing within Spark.
- **Docker/Docker Compose:** Essential for containerization and orchestration of all services in this local development environment, ensuring isolated, reproducible, and portable environments.
- **FastAPI:** A modern, high-performance web framework for building data ingestion APIs with Python 3.7+, offering automatic interactive documentation (Swagger UI). It acts as a Kafka producer.
- **Grafana Alloy:** An OpenTelemetry Collector distribution that is highly configurable and optimized for collecting, processing, and exporting telemetry data (metrics, logs, traces). It acts as a central hub for observability data.
- **Grafana:** An open-source platform for interactive data visualization and monitoring. It is used to create dashboards and visualize metrics and traces.
- **MinIO:** An open-source object storage server that is compatible with Amazon S3 APIs. It simulates an S3-compatible data lake locally.
- **MongoDB:** A popular open-source NoSQL document database. It provides flexible storage for semi-structured data.
- **OpenMetadata:** An open-source metadata management platform that provides a unified data catalog, data lineage, and data quality capabilities, enabling data discovery and governance.
- **OpenTelemetry:** A set of open-source tools, APIs, and SDKs that standardize the collection and export of telemetry data (metrics, logs, and traces) from software applications.
- **PostgreSQL:** A powerful, open-source object-relational database system, serving as a robust SQL datastore for structured data, reference data, and the metadata database for Apache Airflow.
- **Python:** The primary programming language for all ETL pipelines, APIs, scripting, and machine learning components.

- **Spline:** An open-source tool specifically designed for automated data lineage tracking within Apache Spark jobs. It captures metadata about Spark transformations and provides a UI for visualizing data flow.

4.3. Decision Frameworks for Technology Choices

Choosing the right tool for the job is critical. Here, we present frameworks to guide your architectural decisions.

4.3.1. ETL vs. ELT

The choice between ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) depends on your specific needs regarding data volume, latency, and team capabilities.

Feature / Criteria	**ETL (Extract, Transform, Load)**	**ELT (Extract, Load, Transform)**
Data Volume	Better for smaller, more controlled datasets	Ideal for large, unbounded datasets (petabytes to exabytes)
Latency Needs	Typically batch-oriented, higher latency acceptable	Suited for real-time or near real-time, lower latency required
Transformation Logic	Performed before loading into target; requires dedicated staging area	Performed after loading into the data lake; leverages lakehouse compute
Tooling	Traditional ETL tools (Talend, Informatica), custom scripting	Cloud data warehouses (Snowflake, BigQuery), Spark, Databricks, Glue
Team Skills	May lean towards SQL/ETL tool expertise	Strong programming (Python/Scala) and distributed systems knowledge
Cost Model	Fixed infrastructure for ETL tools; less flexible scaling	Scalable compute (Spark, cloud DWs); compute cost scales with usage
Schema Flexibility	Schema-on-write, stricter schema enforcement	Schema-on-read, more flexible, handles schema evolution better

Recommendation: For modern enterprise data platforms dealing with diverse and high-volume data, **ELT with a data lakehouse (like Delta Lake + Spark)** is generally preferred due to its scalability, flexibility, and ability to handle both batch and streaming workloads efficiently. ETL still holds value for highly structured, pre-defined integrations into traditional data warehouses.

4.3.2. Messaging Queues: Kafka vs. Kinesis vs. Pub/Sub

Choosing a streaming platform depends on your operational overhead tolerance, specific features, and cloud strategy.

Feature / Criteria	Apache Kafka (Self-Managed)	AWS Kinesis	Google Cloud Pub/Sub
Throughput/Scale	Extremely high; scales horizontally with brokers and partitions	High; scales with shards	Very high; scales automatically
Operational Overhead	High (requires managing Zookeeper, brokers, maintenance)	Medium (managed service, but shard management is manual)	Low (fully managed, serverless, no infrastructure to manage)
Cloud Lock-in	Low (open-source, portable across clouds/on-prem)	High (AWS-specific)	High (Google Cloud-specific)
Pricing Model	Infrastructure costs + operational expertise	Per shard-hour, data transfer, and data ingested/egressed	Per message operation (publish/subscribe), data transfer
Feature Set	Rich ecosystem (Kafka Connect, Streams API); flexible	Data Firehose (integrations), Data Analytics (real-time SQL)	Global access, automatic scaling, robust IAM integration
Use Case Sweet Spot	Hybrid/multi-cloud, complex streaming apps, full control needed	AWS-native streaming, tight integration with other AWS services	Google Cloud-native, event-driven architectures, simple messaging

Recommendation: For a local development environment, **Apache Kafka** is chosen due to its open-source nature, comprehensive feature set, and high relevance in the industry, which prepares engineers for diverse production environments. For cloud deployments, the choice shifts based on your primary cloud provider and operational preferences. For detailed cloud migration strategies for messaging queues, refer to the **Cloud Migration + Terraform Snippets Deep-Dive Addendum**.

4.3.3. Distributed Processing: Spark vs. Glue/EMR vs. Flink

Choosing a distributed processing engine depends on your workload (batch vs. streaming), cost model, and management preference.

Feature / Criteria	Apache Spark (Self-Managed/Open Source)	AWS Glue (Serverless ETL)	Amazon EMR (Managed Clusters)	Apache Flink (Stream Processing)

Workload Focus	Both Batch & Streaming (Structured Streaming)	Primarily Batch ETL, can do micro-batch streaming	Both Batch & Streaming (various engines)	Primarily Real-time Streaming
Cost Model	Infrastructure + operational overhead; flexible	Serverless, pay-per-use (DPUs/duration)	Instance-based pricing, cluster management costs	Infrastructure + operational overhead; flexible
Resource Isolation	Manual configuration per job/cluster	Automatic, jobs are isolated	Cluster-level isolation	Fine-grained resource control, low-latency stateful processing
Management Overhead	High (setup, maintenance, scaling)	Low (fully managed, no servers to provision)	Medium (managed, but cluster configuration and lifecycle remain)	High (setup, maintenance, scaling)
Development Experience	PySpark/Scala/Java; high control, rich APIs	PySpark/Scala; managed environment; integrates with Data Catalog	PySpark/Scala/Java; integrates with other AWS services	Java/Scala; complex stateful processing APIs, event-time processing
Use Case Sweet Spot	Diverse workloads, fine-grained control, on-prem/hybrid cloud	Ad-hoc ETL, event-driven jobs, data lake transformations	Big data analytics, ad-hoc queries, transient/long-running clusters	Real-time dashboards, fraud detection, complex event processing

Recommendation: For the local environment, **Apache Spark** is chosen because it offers flexibility for both batch and streaming, a rich PySpark API, and a broad industry presence. This provides a strong foundation for understanding distributed processing patterns before transitioning to managed cloud services. In the cloud, the choice shifts based on whether serverless ETL (Glue) or more controlled cluster management (EMR) is preferred for Spark workloads, or if pure low-latency stream processing (Flink) is the priority. For conceptual Terraform snippets for Glue and EMR, refer to the **Cloud Migration + Terraform Snippets Deep-Dive Addendum**.