U.S. Power Plant Net Generation Visualization

Software Architecture Document
Version 1.0

Created by: Tushar Badhe

Date: 24 May 2025

Table of Contents

Introduction

The purpose of this document is to describe the architecture of the U.S. Power Plant Net Generation Visualization solution. It outlines the logical & physical architecture, technology choices, and deployment strategy to meet the specified functional and non-functional requirements. This document serves as a guide for development, deployment, and ongoing maintenance of the system.

Scope

The scope of this document covers the design and implementation of a system that ingests U.S. power plant net generation data from CSV files stored in S3-compatible object storage, transforms and stores this data, exposes it via an API, and visualizes it through a simple web-based user interface. The solution will be containerized using Docker and Docker Compose.

Overview

The solution consists of three main components: a data ingestion and processing backend, a RESTful API, and a web-based user interface. Data will be sourced from CSV files, processed, and stored in a persistent data store. The API will provide endpoints for data retrieval, and the UI will enable users to interactively explore the data. All components will be containerized for easy deployment and management.

Architectural Representation

This document will present the architecture using several views:

- Conceptual View: A high-level overview of the system's main components and their interactions.
- Logical View: A more detailed breakdown of the system's modules, their responsibilities, and the relationships between them.
- Physical View: Illustrates the deployment of software components onto hardware or virtual machines.
- Deployment Strategy: Describes how the solution will be deployed and managed.

Definitions, Acronyms, and Abbreviations

- API: Application Programming Interface
- AWS S3: Amazon Web Services Simple Storage Service
- CSV: Comma Separated Values
- Docker: A platform for developing, shipping, and running applications in containers.

- Docker Compose: A tool for defining and running multi-container Docker applications.
- EPA: Environmental Protection Agency
- eGRID: Emissions & Generation Resource Integrated Database
- FastAPI: A modern, fast (high-performance) web framework for building APIs with Python.
- GenAI: Generative Artificial Intelligence
- MinIO: An open-source, S3-compatible object storage server.
- NFRs: Non-Functional Requirements
- Object Storage: A computer data storage architecture that manages data as objects.
- ORM: Object-Relational Mapping
- RESTful API: An API that conforms to the principles of REST (Representational State Transfer).
- S3: Simple Storage Service (referring to the protocol and compatible services).
- UI: User Interface
- U.S.: United States

References

- EPA eGRID 2023 dataset: https://www.epa.gov/system/files/documents/2025-01/egrid2023_data_rev1.xlsx
- FastAPI Documentation: https://fastapi.tiangolo.com/
- React Documentation: https://react.dev/
- MinIO Documentation: https://min.io/docs/
- Docker Documentation: https://docs.docker.com/

Requirements

Strategic Goal

The strategic goal of this project is to provide an efficient and user-friendly solution for visualizing U.S. power plant net generation data, enabling stakeholders to quickly identify top-performing plants and analyze generation trends by state.

Functional Requirements

The solution shall:

- FR-1: Data Ingestion: Ingest CSV files from an S3-compatible object storage (e.g., MinIO, AWS S3).
    - FR-1.1: Support CSV files conforming to the GEN23 sheet structure of the EPA's eGRID 2023 dataset.

  - FR-1.2: Only support CSV format for ingestion.
- FR-2: Data Transformation and Storage:
  - FR-2.1: Extract relevant fields from the ingested CSV data.
  - FR-2.2: Normalize and clean the data as needed (e.g., handling missing values, standardizing formats).
  - FR-2.3: Store the transformed data persistently.
- FR-3: Data Exposure (API): Provide a RESTful API with the following endpoints:
  - FR-3.1: Retrieve the top N power plants by net generation, ordered by net generation in descending order.
  - FR-3.2: Filter the top N power plants by U.S. state.
  - FR-3.3 (Bonus): Implement basic token-based authentication for API access.
- FR-4: User Interface: Provide a web-based interface that allows users to:
  - FR-4.1: Select a U.S. state from a dropdown or similar control.
  - FR-4.2: Specify the number of top plants (N) to view via an input field.
  - FR-4.3: Display the results (top N plants for the selected state) in a clear table or chart.
- FR-5: Containerization:
  - FR-5.1: Containerize all solution components using Docker.
  - FR-5.2: Provide a `docker-compose.yml` file for simplified local setup and orchestration.

Features

- Data Ingestion: An manually triggered process for reading CSV files from object storage.
- Data Processing: Logic for parsing, validating, transforming, and storing data.
- RESTful API: For programmatic access to cleaned and aggregated power plant data.
- Interactive Web UI: For visual exploration of the data.
- Containerized Environment: Ensures portability and ease of deployment.

Persona

- Data Analyst: A user interested in quickly identifying top power plants by net generation within specific states. They will use the UI to explore trends and gather insights.
- Developer: A user who needs to integrate with the power plant data. They will primarily use the API to retrieve information.
- System Administrator: Responsible for deploying, monitoring, and maintaining the solution.

Non-Functional Requirements

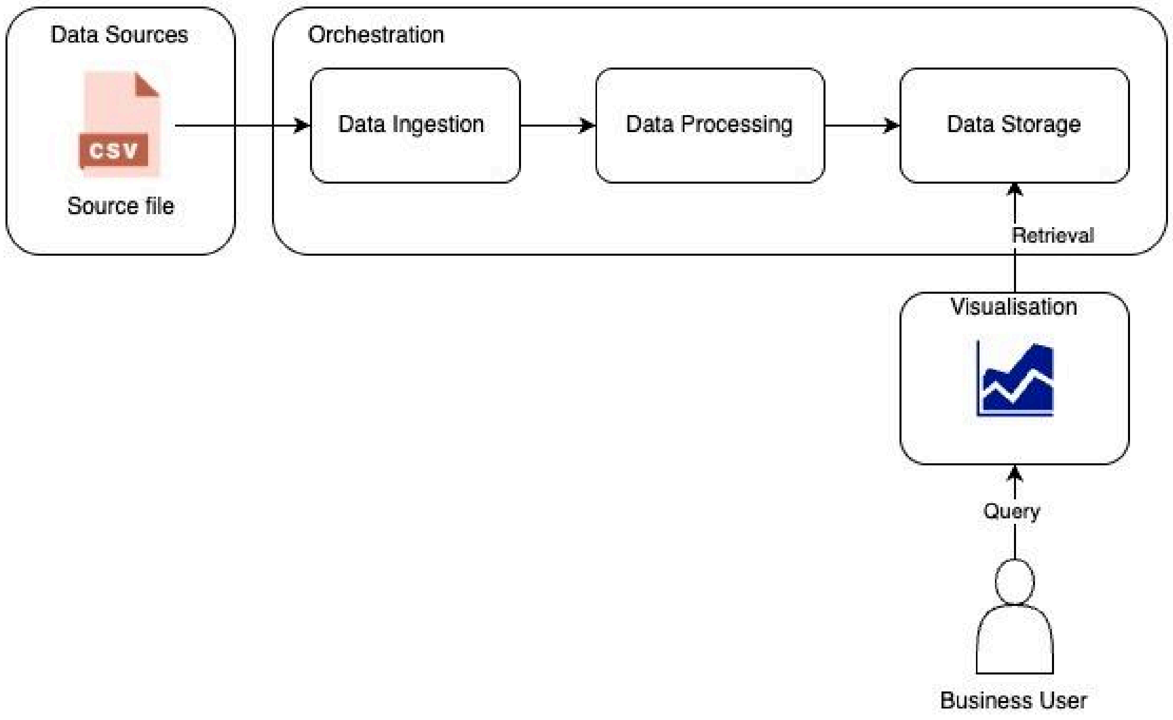
- NFR-1: Scalability: The solution should be designed to handle an increasing number of CSV files and growing data volumes. The API should be able to support multiple concurrent users without significant performance degradation.
- NFR-2: Maintainability: The codebase should be clean, modular, and well-documented to facilitate future enhancements and bug fixes.
- NFR-3: Security (Bonus): API endpoints should be protected by basic token-based authentication.
- NFR-4: Usability: The UI should be intuitive and easy to navigate for users with varying technical backgrounds.
- NFR-5: Observability: The system should provide mechanisms for monitoring its health, performance, and data processing status (e.g., logs, metrics).
- NFR-6: Portability: The containerized solution should be easily deployable across different environments (local, various cloud providers).

Architectural Context

The solution operates within a typical web application context, leveraging cloud-native principles through containerization. It interfaces with external object storage for data input and provides a web-based interface for user interaction

Conceptual View

The conceptual view provides a high-level overview of the major components and their interactions.



1. Data Sources:
    - Source file (CSV): This represents the origin of the raw data. In the context of a power plant, this could be various operational data such as sensor readings (temperature, pressure, flow rates), electricity generation metrics, fuel consumption, maintenance logs, environmental data, market prices, or even weather data. The `.csv` (Comma Separated Values) format suggests that the initial data is typically structured and often tabular.

2. Orchestration:

This section represents the core data pipeline responsible for transforming raw data into actionable insights.

- Data Ingestion:
    - Purpose: This is the first step in the pipeline where raw data is collected and brought into the system from the data sources.
    - Process: For a power plant, this would involve reading GEN23.csv file and performing initial validation checks.
    - Output: The ingested raw data, ready for further processing.
- Data Processing:
    - Purpose: To clean, transform, enrich, and aggregate the ingested data to make it suitable for analysis and storage.
    - Process: In a power plant context, this could involve:
        - Cleaning: Handling missing values, correcting errors, removing duplicates.
        - Missing values: Handling missing values
        - Duplicate rows: Removing duplicate rows
        - Normalization/Standardization: Ensuring data standardisation
    - Output: Processed and refined data.
- Data Storage:
    - Purpose: To persistently store the processed data in a structured and organized manner, optimized for retrieval and analysis.
    - Types: This could be various types of file storage or databases or data warehouses depending on the data volume, velocity, and variety:
        - Object storage : For storing file objects
        - Relational Databases (e.g., SQL Server, PostgreSQL): For structured historical data.
        - NoSQL Databases (e.g., MongoDB, Cassandra): For large volumes of unstructured or semi-structured data, or for high-velocity data.
        - Data Lake: For storing raw data in its native format, often in cloud storage (e.g., AWS S3, Azure Data Lake Storage).
        - Data Warehouse (e.g., Snowflake, BigQuery): Optimized for analytical queries and reporting.
    - Process: The processed data is loaded into the chosen storage system, often in a way that supports efficient querying.
    - Retrieval: The "Retrieval" arrow indicates that data can be accessed from Data Storage for various purposes, including visualization.

3. Visualization:

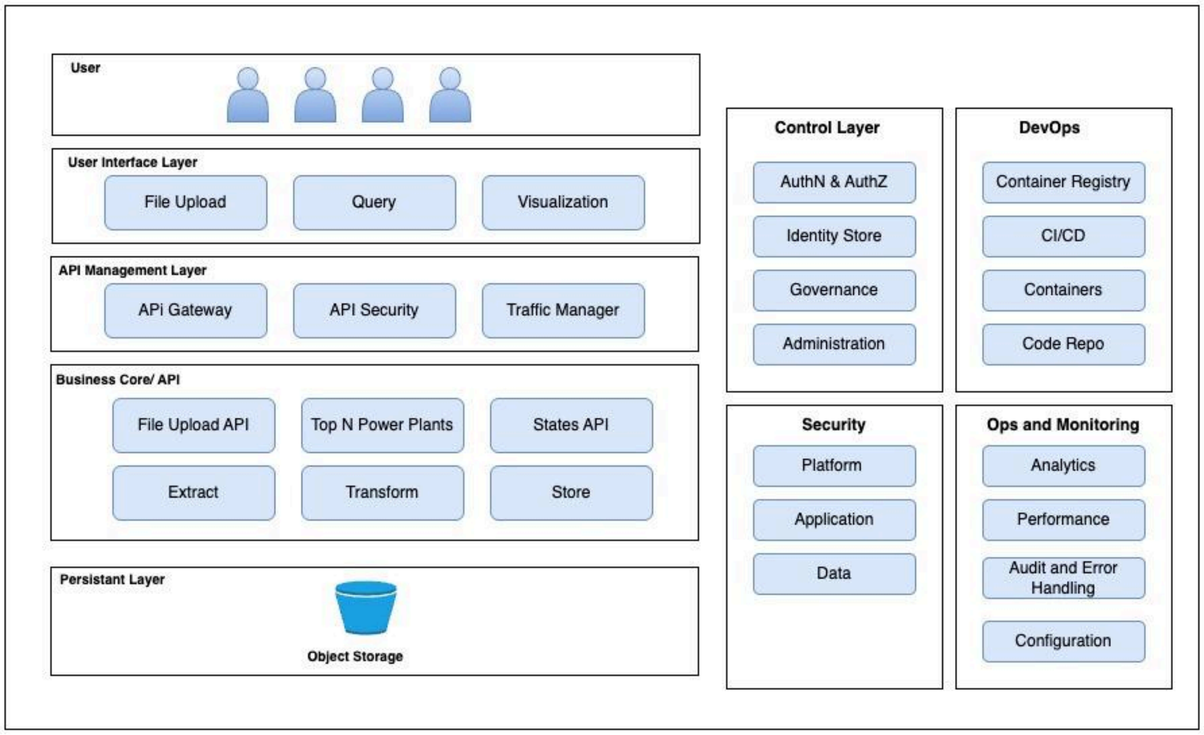
- Purpose: To present complex data in an easily understandable and insightful graphical format.
- Process: A visualization tool (represented by the graph icon) queries the processed data from Data Storage.
- Output: Dashboards, reports, charts (e.g., line graphs for trends, bar charts for comparisons, heat maps for operational performance), and other visual representations that provide insights into power plant operations.

4. Business User:

- Purpose: The end-user who interacts with the system to gain insights and make informed decisions.
- Interaction: Business User interacts with the Visualization layer to retrieve specific information.

Logical View

The logical view breaks down the system into its main modules, showing their responsibilities and interactions. Layered architecture provides separation of concerns at various layers allowing application maintainability and scalability easy.



1. User Layer:

- Users: These are the human individuals who interact with the system. They could be power plant operators, business analysts, engineers, administrators, or managers. Their activities range from uploading data to querying information and visualizing results.

2. User Interface Layer:

- Purpose: This layer provides the graphical interface through which users interact with the system.
- Components:
    - File Upload: Allows users to manually upload data files (e.g., CSVs, historical logs) into the system. This is crucial for integrating data that might not be automatically ingested.
    - Query: Enables users to define and execute queries against the stored data to retrieve specific information. This could involve complex data filtering and selection.
    - Visualization: Presents data in graphical formats (charts, dashboards, maps) to help users understand trends, anomalies, and performance metrics. This is the output of the data processing.

3. API Management Layer:

- Purpose: This layer manages how external systems and internal components communicate with the core business logic, ensuring security, control, and efficiency.
- Components:
    - API Gateway: Acts as a single entry point for all API calls. It handles request routing, composition, and protocol translation. It's essential for microservices architectures.
    - API Security: Implements authentication, authorization, encryption, and other security measures for APIs to protect data and resources.
    - Traffic Manager: Manages and directs network traffic to various services, optimizing performance, ensuring high availability, and potentially handling load balancing.

4. Business Core/API Layer:

- Purpose: This is the heart of the application, containing the core business logic and processing capabilities specific to power plant operations and data. It exposes functionalities as APIs.
- Components:
    - File Upload API: An API endpoint specifically designed to receive and process uploaded files, likely feeding them into the data ingestion pipeline.
    - Get Plants API: Suggests an API to retrieve or analyze data related to the top 'N' performing power plants based on certain criteria (e.g., efficiency, generation, uptime). This implies comparative analysis capabilities.
    - Get States API: An API allows to retrieve the states within the United states.
    - Extract: A module responsible for extracting data from file sources. This is part of the "ETL" (Extract, Transform, Load) process.
    - Transform: A module that cleans, validates, aggregates, and transforms the extracted raw data into a format suitable for analysis and storage.
    - Store: A module responsible for writing the processed and transformed data into the persistent storage layer.

5. Persistent Layer:

- Purpose: This layer is responsible for the long-term storage of all data processed and managed by the system.
- Component:
    - Object Storage (s3-compatible): A highly scalable, durable, and available storage service for unstructured data, compatible with Amazon S3. This is ideal for storing large volumes of data like raw sensor readings, processed data files, or even backups. Its "s3-compatible" nature suggests cloud-native or cloud-like infrastructure.

6. Control Layer:

- Purpose: This layer focuses on managing access, identity, and governance within the system.
- Components:
    - AuthN & AuthZ (Authentication & Authorization):
        - Authentication (AuthN): Verifies the identity of users or systems (e.g., username/password, API keys, tokens).
        - Authorization (AuthZ): Determines what authenticated users or systems are allowed to do (e.g., read-only access, write permissions).

    - Identity Store: A database or directory service that stores user identities, roles, and permissions (e.g.,Social media authentication, Active Directory, a dedicated identity management system).
    - Governance: Establishes policies, rules, and processes to ensure compliance, data quality, security, and ethical use of the system and its data.
    - Administration: Provides tools and interfaces for managing the system itself, including user management, configuration changes, system monitoring setup, etc.

7. DevOps:

- Purpose: This section represents the practices, tools, and processes that enable rapid and reliable software delivery and operation.
- Components:
    - Container Registry: A centralized repository for storing and managing Docker images or other container images.
    - CI/CD (Continuous Integration/Continuous Deployment): Automated pipelines for building, testing, and deploying code changes frequently and reliably.
    - Containers: Lightweight, portable, and self-sufficient units that encapsulate an application and its dependencies, ensuring consistent execution across different environments(e.g., Docker, Kubernetes).
    - Code Repo (Code Repository): A version control system (e.g., Git) where all the source code for the application is stored and managed.

8. Security:

- Purpose: This section outlines the different levels at which security measures are applied.
- Components:
    - Platform: Security applied at the infrastructure level (e.g., network security, server hardening, cloud security configurations).
    - Application: Security measures within the application code itself (e.g., secure coding practices, input validation, vulnerability scanning).
    - Data: Security related to the data itself (e.g., encryption at rest and in transit, data masking, access control on data stores).

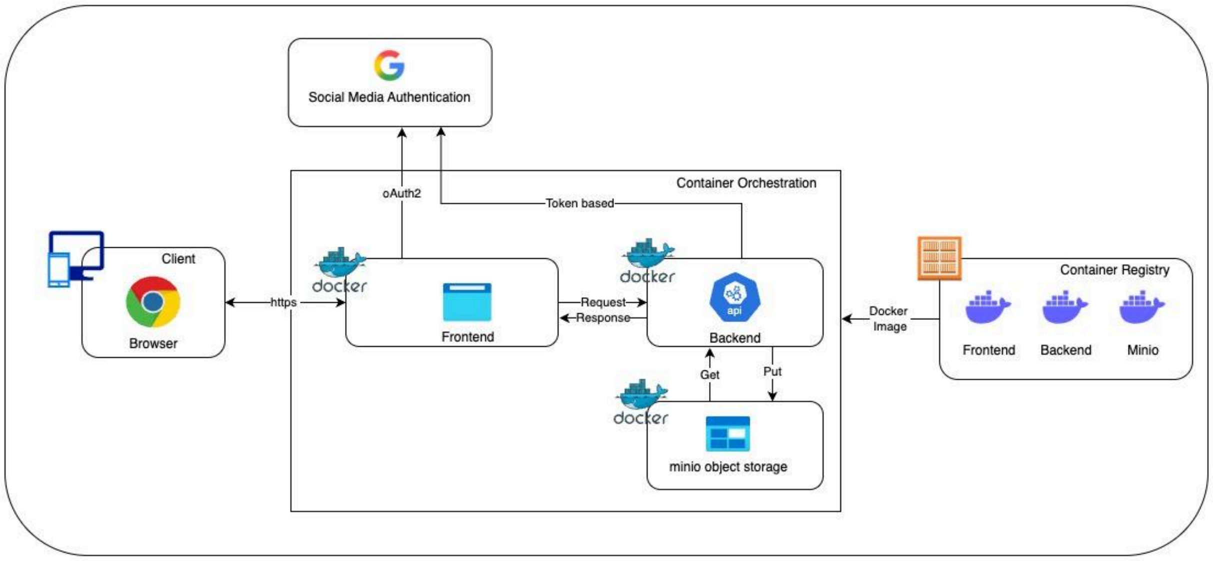
9. Ops and Monitoring:

- Purpose: This section covers the operations and monitoring aspects, ensuring the system runs smoothly and efficiently.
- Components:
    - Analytics: Tools and processes for analyzing operational data (logs, metrics) to gain insights into system performance, usage patterns, and potential issues.
    - Performance: Monitoring and optimizing the system's speed, responsiveness, and resource utilization.
    - Audit and error handling: Mechanisms for logging system events, user actions (auditing), and effectively handling, logging, and notifying about errors and exceptions.
    - Configuration: Management of system settings, parameters, and environment variables across different components and environments.

Overall System Purpose and Interdependencies:

This high-level view depicts a robust, scalable, and secure data platform for a power plant. It starts with user interaction for data input and querying, processes data through a managed API layer and core business logic, stores it persistently, and is supported by critical functions like security, identity management, DevOps for continuous delivery, and comprehensive operations/monitoring.

Deployment View



Deployment view captures runtime components, illustrates communication paths, highlights deployment technologies, indicates physical/logical grouping

1. Client:

- Browser: Users interact with the application through a web browser on various devices (desktop, mobile, tablet, indicated by the icons).
- HTTPS: Communication between the client and the Frontend is secured using HTTPS.

2. Authentication:

- Social Media Authentication (Google): The application integrates with Google for social media authentication.
- OAuth2: This protocol is used for authorizing the application to access user data from Google after successful authentication.
- Token-based Authentication: After OAuth2 authentication, a token is issued to the client, which is then used for subsequent authenticated requests to the backend.

3. Application Core (Container Orchestration):

This section represents the heart of the application, deployed within a container orchestration environment (e.g., Kubernetes, Docker Swarm, docker desktop (local) etc.)

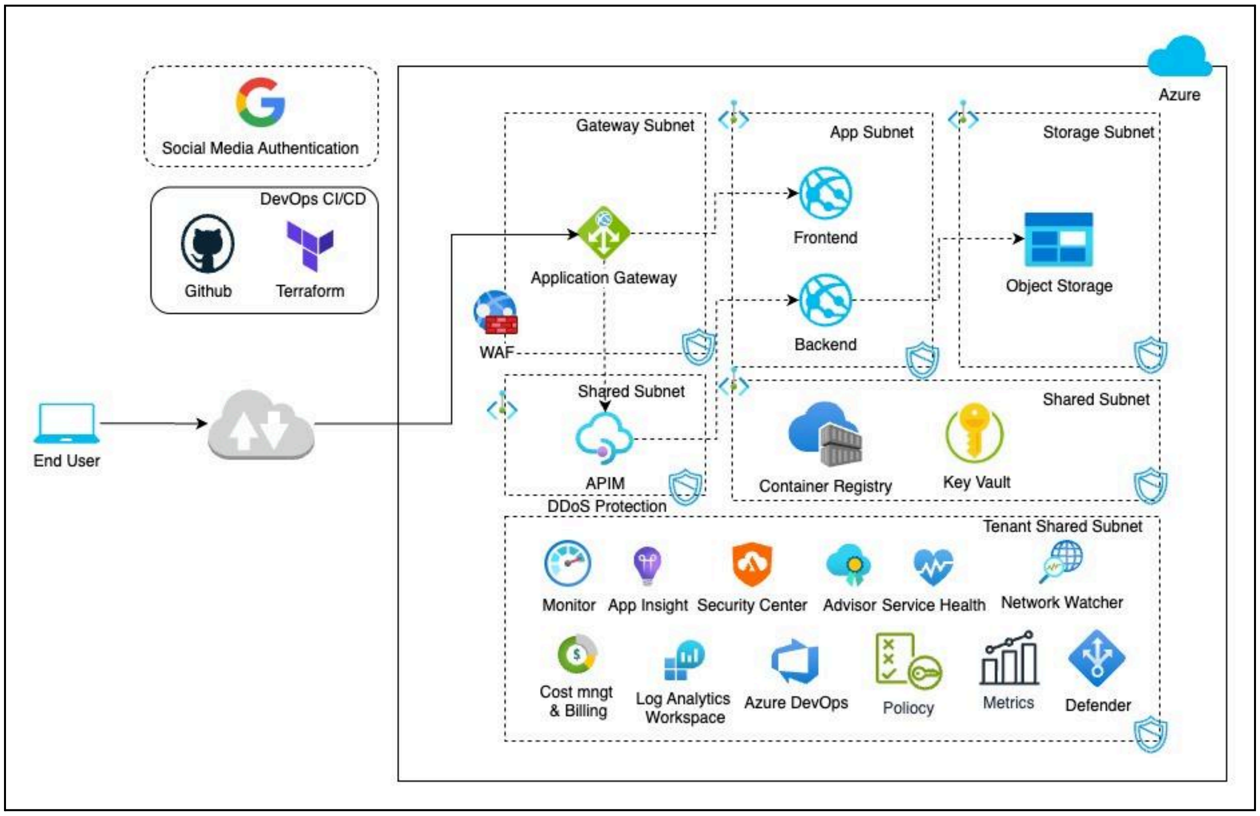
- Frontend:
    - This component serves the user interface to the client's browser.
    - Frontend application is developed in react ts
    - It's containerized using Docker.
    - It communicates with the Backend for data and application logic.
- Backend:
    - This component houses the core application logic and APIs.
    - FASTAPI exposes the RESTFul api
    - It's containerized using Docker.
    - It handles requests from the Frontend and processes them.
    - It interacts with Minio Object Storage for data persistence.
- Minio Object Storage:
    - This component provides object storage capabilities, likely used for storing visualizations, power plan data, or other static assets.
    - It's containerized using Docker.
    - The Backend uses "Get" and "Put" operations to interact with Minio, indicating data retrieval and storage.

4. Container Registry:

- This is a central repository for storing Docker images of the application's components.
- It contains images for the Frontend, Backend, and Minio, making them readily available for deployment and scaling by the container orchestration system.

Infrastructure View

The Infrastructure view illustrates how the logical components are mapped to physical deployment units (containers in this case).



PowerPlan Visualization Application, primarily built on Microsoft Azure. It can be hosted on any cloud provider. However, Azure is being used as, AIQ uses Azure extensively.It showcases a modern, cloud-native design with a strong emphasis on security, scalability, and operational efficiency.

Key components are as follows

I. User Access and Authentication:

- End User: Represents the consumers of the application.
- DDoS Protection: All incoming traffic is protected against Distributed Denial of Service (DDoS) attacks, ensuring application availability.

- Social Media Authentication (Google): Users authenticate to the application using their Google social media accounts, leveraging existing identity providers for simplified access.

II. DevOps and CI/CD:

- DevOps CI/CD: The development and deployment pipeline is automated using Continuous Integration/Continuous Delivery (CI/CD) practices.
    - Github: Likely used for source code management and version control.
    - Terraform: Employed for Infrastructure as Code (IaC), enabling automated provisioning and management of Azure resources.

III. Azure Networking and Application Delivery:

- VNet: The entire infrastructure is hosted within Microsoft Azure VNet.
- Gateway Subnet:
    - Application Gateway: Acts as a web traffic load balancer that enables you to manage traffic to your web applications. It routes requests to the appropriate backend services (Frontend).
    - WAF (Web Application Firewall): Provides centralized protection of your web applications from common exploits and vulnerabilities. It's integrated with the Application Gateway.
- Shared Subnet:
    - APIM (API Management): Manages and publishes APIs, handling aspects like security, throttling, and analytics. It likely exposes APIs consumed by the Frontend.

IV. Application Subnets and Components:

- App Subnet:
    - Frontend: The user-facing part of the application, responsible for rendering the visualization and interacting with the backend. Hosted using AppServices. App services is a PaaS solution by Azure which helps to host docker application with no time, taking care of scaling, r-b deployment and many other features.
    - Backend: The core logic of the application, processing data and interacting with the storage layer. Backed is also hosted inside AppService
- Shared Subnet (within App Subnet scope):
    - Container Registry: Stores and manages Docker container images for the application components (Frontend, Backend, potentially others).
    - Key Vault: Securely stores and manages sensitive information such as API keys, passwords, certificates, and encryption keys used by the application components.

V. Storage Subnet:

- Storage Subnet:

  - Object Storage: Likely Azure Blob Storage, used for storing large amounts of unstructured data such as raw PowerPlan data, visualization assets, or application logs.

VI. Tenant Shared Subnet (Shared Services/Monitoring/Security):

This subnet houses a suite of shared Azure services crucial for monitoring, security, governance, and cost management. These services are leveraged across the entire application and potentially other applications within the same Azure tenant.

- Monitor: Azure Monitor provides comprehensive monitoring of Azure resources, applications, and operating systems.
- App Insights (Application Insights): An Application Performance Management (APM) service that monitors live web applications, detects performance anomalies, and helps diagnose issues.
- Security Center: Provides unified security management and advanced threat protection across hybrid cloud workloads.
- Advisor: Azure Advisor provides personalized recommendations to optimize cost, performance, reliability, and security of your Azure resources.
- Service Health: Notifies you about Azure service issues, planned maintenance, and health advisories relevant to your resources.
- Network Watcher: Provides tools to monitor, diagnose, and gain insights into network performance and health.
- Cost Management & Billing: Helps you understand and manage your Azure spending and optimize cloud costs.
- Log Analytics Workspace: A centralized repository for collecting, indexing, and analyzing log data from various Azure resources.
- Azure DevOps: While Github and Terraform are shown for CI/CD, Azure DevOps might be used for broader project management, work item tracking, or additional build/release pipelines.
- Policy: Azure Policy helps enforce organizational standards and assess compliance at scale.
- Metrics: Azure Metrics provides quantitative data about your Azure resources.
- Defender: Azure Defender (now Microsoft Defender for Cloud) provides comprehensive threat protection capabilities across your cloud and hybrid environments.

Key Characteristics and Design Principles:

- Cloud-Native: Leverages various Azure services designed for cloud environments.
- Security-First: Multiple layers of security are implemented, including DDoS Protection, WAF, Key Vault, Security Center, and Defender.
- Scalability: Load balancers (Application Gateway), containerization, and the ability to scale Azure services enable the application to handle varying workloads.
- Observability: Comprehensive monitoring (Monitor, App Insights, Network Watcher, Log Analytics, Metrics) ensures visibility into application health and performance.

- Automation: DevOps practices with Github and Terraform enable automated deployments and infrastructure management.
- Centralized Management: Shared services in the Tenant Shared Subnet provide centralized control over security, governance, and cost.

This architecture provides a robust, secure, and scalable foundation for the PowerPlan Visualization Application, enabling efficient development, deployment, and operation within the Azure cloud.

Technology Choices

The following table shows a set of technologies, infrastructure components, platforms, and frameworks chosen along with their rationale.

Component	Technology	Rational
User Interface	React TS	A popular, component-based JavaScript library for building user interfaces. It offers a strong ecosystem, good performance, and a declarative way to manage UI state, making it suitable for interactive web applications.
Backend	python™ FastAPI	Python is a versatile language with a rich data science and backend development ecosystem. FastAPI is chosen for its high performance, automatic interactive API documentation (Swagger UI/ReDoc), asynchronous support, and strong type hinting. It's excellent for building robust and fast RESTful APIs.
Object Storage	MINIO	MinIO provides S3-compatible object storage that can be easily containerized and run locally, making development and testing straightforward without requiring a full cloud

		setup. It offers a seamless transition to AWS S3 and Azure Blob storage in a cloud environment due to API compatibility. This choice provides cloud neutrality at the storage layer.
Conterazation		Docker provides a consistent environment for all application components, abstracting away differences in underlying infrastructure. Docker Compose simplifies the orchestration of multi-container applications, making local setup and development highly efficient.
Infrastructure As Code		Terraform is a widely adopted open-source IaC tool that allows defining and provisioning cloud infrastructure using declarative configuration files. It supports multiple cloud providers (AWS, Azure, GCP), enabling consistent and repeatable deployments.
API Management		Azure APIM serves as an API gateway, managing and publishing APIs. It provides a single entry point for consumers, handling security, throttling, and analytics. This ensures consistent access, scalability, and monitoring of APIs, facilitating their efficient use and management in a cloud environment.
Frontend deployment		App Service in Azure is used for hosting web applications and APIs. It simplifies deployment by supporting containerized applications. App Service handles scaling,

		load balancing, and provides features like auto-scaling and easy deployment. It’s a platform-as-a-service (PaaS) solution, reducing infrastructure management and enhancing development efficiency.
Backend deployment		App Service in Azure is used for hosting web applications and APIs. It simplifies deployment by supporting containerized applications. App Service handles scaling, load balancing, and provides features like auto-scaling and easy deployment. It’s a platform-as-a-service (PaaS) solution, reducing infrastructure management and enhancing development efficiency.
Persistent Layer		Blob storage -Cost-effective, scalable storage for large amounts of unstructured data, like documents, images, and logs.
Control Layer		Google Authentication simplifies user login by leveraging existing Google accounts, reducing the need for new credentials. This enhances user experience, streamlines onboarding, and potentially increases security through Google's robust authentication mechanisms. It also aligns with commonly used authentication patterns for web applications
Security		Azure Web Application Firewall (WAF) protects web applications from common exploits and vulnerabilities like SQL injection

		and cross-site scripting. It provides centralized security, inspecting HTTP traffic, and filtering malicious requests, ensuring application availability and data security. This enhances overall security posture in Azure.
Dev & Automation	GitHub	GitHub repositories are used for source code management and version control. This facilitates collaboration, allows for tracking changes, and enables multiple developers to work on the project concurrently. It's a central location for managing the code base and enabling CI/CD pipelines.
Ops and Monitoring	Azure Monitor	Azure Monitor is used for collecting and analyzing telemetry data from Azure resources and applications. The rationale is to gain insights into performance, health, and usage, enabling proactive issue detection, diagnostics, and resource optimization. It ensures application reliability and provides operational visibility within Azure.
Infrastructure		Azure's global infrastructure and flexible scaling options make it ideal for cloud deployments. Its diverse service offerings, including AI, machine learning, and databases, empower businesses to innovate and make data-driven decisions. Azure also offers robust security and compliance, ensuring data protection

Deployment Strategy

The deployment strategy focuses on containerization and orchestration for both local development and cloud environments.

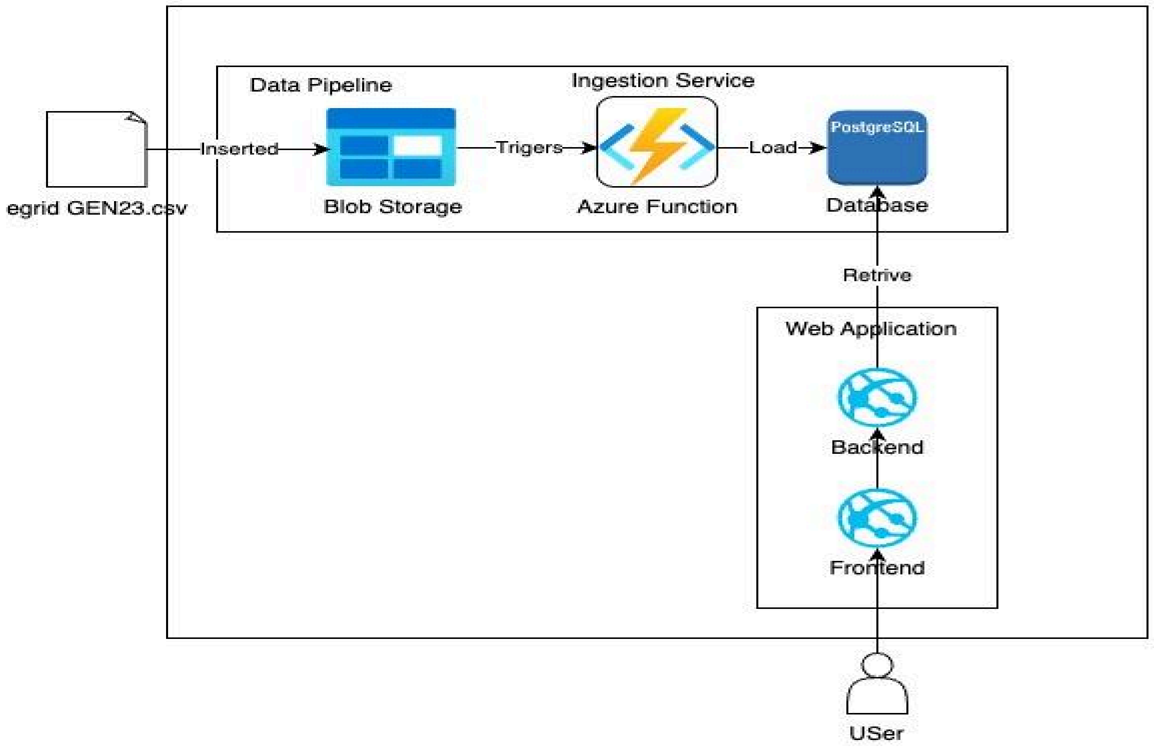
1. Local Development:
    - Docker Compose: The primary tool for local setup. Developers can clone the repository, run `docker-compose up --build`, and have all services running.
    - Hot Reloading: Configure development servers (e.g., React's development server, FastAPI's Uvicorn with reload) within Docker containers to allow for quick iteration.
2. Container Image Management:
    - Dockerfiles will be created for each service (backend, frontend).
    - Images will be built (`docker build`) and can be pushed to a container registry (e.g., Docker Hub, AWS ECR) for cloud deployment.
3. Data Ingestion Trigger:
    - Initial Approach (local): Manual file upload through web application takes care of initial needs of the application to store files in minio s3-compatiable storage.
    - Further enhancement : The `data_ingestion` (independent) service can be configured to poll the MinIO bucket at a set interval for new CSV files. Alternatively, a manual API endpoint on the `backend` can be exposed to trigger an ingestion run, or a simple command can be executed within the `data_ingestion` container.
    - Cloud (Azure): Azure blob event notification can be configured to trigger Azure function which will process the file and store the data into blob storage
    - Cloud (AWS): An S3 event notification can be configured to trigger an AWS Lambda function, which in turn can invoke the `data_ingestion` service (e.g., by sending a message to SQS that the service consumes, or directly triggering a Fargate task).
4. Database Migrations (further enhancement):
    - Currently, databases are not being used for simplicity and considering local development with single user access. However, databases can be introduced to store the process data in structured form for easy retrieval across multiple user workload.
    - Database schema changes will be managed using an ORM migration tool (e.g., Alembic with SQLAlchemy).
    - Migration scripts will be run as part of the `backend` or `data_ingestion` service startup (in development) or as a separate step in CI/CD pipelines (in production).

5. CI/CD Pipeline (Cloud):
    - Automate testing, building Docker images, and pushing them to a container registry upon code commits.
    - Automate deployment to cloud environments using Terraform to provision infrastructure and deploy services to Azure Appservice, Azure container Apps, ACI, or AKS. In AWS, ECS/EKS could be good choice for container orchestration.
6. Monitoring:
    - Logging: Centralized logging using tools like ELK Stack (Elasticsearch, Logstash, Kibana) or cloud-native solutions (Azure:Application insights, log analytics workspace; AWS:CloudWatch Logs). All services will log structured output.
    - Metrics: Collect application-level metrics (e.g., API response times, number of ingested records, errors) using Prometheus/Grafana or cloud-native monitoring services (Azure Monitor; AWS CloudWatch Metrics). FastAPI's Prometheus integration can be used.
    - Health Checks: Configure health check endpoints for each service (`/health`) to allow orchestrators (Docker Compose, ECS) to monitor their status.
    - Alerting: Set up alerts based on critical metrics or log patterns (e.g., high error rates, service down).

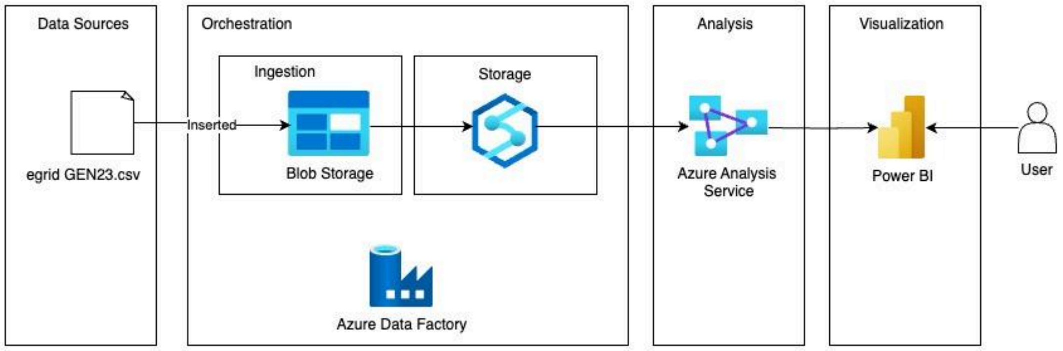
Changing Non-Functional Requirements

The modular and containerized architecture provides flexibility to address changing NFRs:

- Scalability:
    - Increased Data Volume/Ingestion Rate:
        - Data Ingestion: For higher ingestion throughput, scale out the `data_ingestion` service instances. As mentioned below, Ingestion service should be separated. Once a file gets uploaded into blob, blob triggers an event which will invoke Azure function. Azure function performs ETL operations and stores data into relational databases i.e. postgres. Web application implements complex retrieval queries. Scalability is achieved by event driven ingestion, serverless processing, decoupling, scalable web application



- **Further enhancement**, Ingestion pipeline can be further enhanced using Enterprise Azure data factory as shown below



      - Azure-based data analytics and business intelligence (BI) pipeline is designed for scenarios where raw data needs to be ingested, processed, stored in an optimized format for analytics, and then visualized for end-users to gain insights. Solution is useful for building a modern data warehouse, lakehouse from various data sources into a scalable data lake. It supports BI tools like power BI, enabling users to create dashboard, reporting, and ad-hoc analysis on data.
      - Advantage of the solution is that it allows business users to independently explore analysed data (with analysis service)
        - Informed decision making
        - HIstorical data analysis and Analytics
        - Fueling GenAI based use cases with data
        - Highly scalable and elastic solution with ADF, Data lakes, analysis service, and power BI
        - Performance optimization for analytics
        - Data governance and management
        - SImplified development and integration
        - Flexibility and extensibility
    - Database: Introduction of database for storing processed data for faster retrieval. It also allows server complex user queries. Azure database for Postgres would be a wise option. Utilize PostgreSQL scaling options (e.g., vertical scaling to a larger instance, read replicas for the API if reads dominate, sharding if data grows excessively large and query patterns allow).
  - Increased web request
    - Web application (frontend and backend) scalability
      - AppService: App service is a fully managed platform for building, deploying and scaling web applications, web api, and backend.
        - Horizontal scaling (scale-out): Primary way to handle scalability. Manually setting a fixed number of instances or autoscale based on CPU, memory etc.
        - Vertical scaling: by increasing the size of VM which runs app services.
    - Application Gateway
      - Layer 7 load balancer, distributes incoming web traffic into backend services. Auto scaling allows automatic scale out based on traffic load. When scaled down, the application gateway gracefully drains connections.

- Increased API Traffic:
  - APIM
    - Acts as an API gateway that publishes, secures, transforms, maintains, and monitors APIs and provides single consistent entry point for all API consumers
    - Scaling mechanisms : tier based scaling, scaling units, auto scaling rules, throttling, rate limit and multi-region deployment

- Performance:
  - API Response Times:
    - Database Indexing: Optimize database queries by adding appropriate indexes to frequently queried columns (e.g., `state`, `net_generation`).
    - Caching: Implement caching for frequently requested data (e.g., top N plants for common states) at the API level (e.g., using Redis) or even at the database level.
    - Query Optimization: Refine SQL queries in the `ServiceLayer` to be more efficient.
  - Data Ingestion Speed:
    - Event driven approach: Async events and serverless Azure function is designed to handle the performance needs of the application
    - Batch Inserts: Optimize `DataLoader` to perform bulk inserts rather than row-by-row inserts for better database performance.
    - Parallel Processing: If CPU-bound, process multiple CSV files or chunks of a large CSV in parallel (e.g., using Python's `multiprocessing` or distributed task queues like Celery).
    - Stream Processing: Process CSV data in streams to reduce memory footprint for very large files.
- Reliability/Resilience:
  - Redundancy**:** Load balancing across multiple instances of Frontend, Backend, Application Gateway, and APIM eliminates single points of failure.
  - Managed Services**:** Azure services like Blob Storage, Key Vault, and Container Registry offer inherent high availability and data durability.
  - Network Protection**:** DDoS Protection and WAF safeguard against attacks that could impact availability.

    - Proactive Monitoring: Azure Monitor, Application Insights, and Log Analytics enable real-time health monitoring and issue detection, ensuring rapid response and continuous operation.
    - Fault Tolerance: Docker Compose provides basic restart policies. In a cloud environment, Appservices ensure fault tolerance
    - Error Handling: Robust error handling and logging at all layers to identify and debug issues quickly.
- Security:
    - Authentication (Token-based): Implement JWT (JSON Web Token) based authentication.
        - A login endpoint will issue a token upon successful credentials validation.
        - API endpoints will require a valid JWT in the `Authorization` header.
        - `python-jose` for JWT encoding/decoding and `passlib` for secure password hashing.
    - Authorization: If different user roles are needed, role-based access control (RBAC) could help
    - Secrets Management: Use environment variables for sensitive information in development. In production, leverage cloud secret managers (e.g., azure key vault, AWS Secrets Manager, HashiCorp Vault) or Kubernetes Secrets.
    - Network Security: Configure network security groups/firewalls to restrict traffic between services and only expose necessary ports to the internet.
- Maintainability:
    - Code Quality: Adherence to coding standards (PEP 8 for Python), clear comments, and robust unit/integration tests.
    - Modular Design: The chosen architecture promotes clear separation of concerns (frontend, backend, data ingestion), simplifying maintenance.
    - API Versioning: If the API evolves significantly, versioning can be implemented(e.g., `/v1/plants`, `/v2/plants`).

Monitoring

Monitoring is crucial for understanding system health, performance, and identifying issues.

1. Logging:
    - Centralized Logging: All services (Backend API, Data Ingestion, Frontend Nginx/server) will be configured to send their logs to a centralized logging system.

        - Local: Docker's logging drivers can direct logs to the console or files. A simple local setup could use `docker logs -f <container_name>`.
        - Cloud: Utilize cloud-native solutions like AWS CloudWatch Logs, Google Cloud Logging, or Azure Monitor Logs. Alternatively, self-host an ELK Stack (Elasticsearch, Logstash, Kibana) or Loki+Grafana.
    - Structured Logging: Emit logs in a structured format to enable easier parsing, filtering, and analysis.
    - Key Information: Logs should include timestamps, service name, log level (INFO, WARN, ERROR), request IDs (for tracing), and relevant messages/error details.
2. Metrics:
    - Application Metrics:
        - FastAPI Backend: Use a library like `fastapi-limiter` for rate limiting (can also expose metrics) or custom Prometheus exporters to track:
            - API request counts, latencies (per endpoint).
            - Error rates (e.g., 5xx errors).
            - Database query times.
            - Number of authenticated vs. unauthenticated requests.
        - Data Ingestion Service: Track:
            - Number of CSV files processed.
            - Number of records ingested/skipped.
            - Ingestion duration per file.
            - Errors during parsing, transformation, or loading.
    - System Metrics:
        - Monitor resource utilization of containers and underlying hosts (CPU, memory, disk I/O, network I/O).
        - Database Metrics: PostgreSQL metrics like active connections, query duration, cache hit ratio, disk usage.
    - Tools:
        - Prometheus & Grafana: A powerful open-source combination for time-series data collection and visualization. Export metrics from services to Prometheus, and visualize in Grafana dashboards.
        - Cloud-Native: AWS CloudWatch Metrics, Azure Monitor, Google Cloud Monitoring for managing metrics in the cloud.
3. Health Checks:
    - Each service will expose a `/health` or `/status` endpoint that returns a 200 OK if the service is operational and can connect to its dependencies (e.g., database, MinIO).

- These endpoints are used by orchestrators (Docker Compose, ECS, Kubernetes) to determine container health and manage restarts.

4. Alerting:
    - Set up alerts based on predefined thresholds for critical metrics or log patterns.
    - Examples:
        - High error rate (e.g., 5% of API requests return 5xx errors).
        - API response times exceeding a threshold (e.g., 1 second).
        - Data ingestion failures or prolonged processing times.
        - Container restarts or unhealthy status.
        - Database disk space nearing capacity.
    - Notification Channels: Integrate with notification services like Slack, email, PagerDuty, or SMS.
5. Dashboarding:
    - Create comprehensive dashboards using Azure Monitor, Grafana or cloud provider's dashboarding tools to visualize key metrics, logs, and health status at a glance.

By combining these monitoring strategies, the solution's operational health and performance can be effectively tracked, allowing for proactive issue resolution and continuous improvement.

Conclusion

This document outlines the architecture for the U.S. Power Plant Net Generation Visualization solution, detailing the design principles, technology choices, deployment strategy, and monitoring plan. The proposed architecture leverages containerization and modular design to ensure scalability, maintainability, and portability. By following this guide, the development team can build and deploy a system that effectively meets the functional and non-functional requirements. This architecture is designed to be adaptable, allowing for adjustments as needs evolve, such as scaling for increased data volume or adding more robust security measures. Continuous monitoring and adherence to best practices will ensure the system's reliability and performance, providing valuable insights into U.S. power plant net generation data.