Ask:

Do we have different roles types?
Internal, external, enterprise vs. consumer
How big are the machine-readable files (MRFs)?

Data Ingestion

1) Raw ingestion into object storage: durable “source of truth” for raw data. We need to ingest large machine-readable files (MRFs) of different formatted hospital/payer negotiated rates.

2) Normalization stage: clean the results into a unified internal schema - nightly rebuild of normalized clean tables and indices. After each pipeline stage (parse successful, normalization table updated) we could trigger an event. Systems downstream can react immediately. Interplay of batch consistency with streaming for fast updates. Use streaming for event flow orchestration (Kafka) and small incremental updates.

Come with tradeoffs - being able to catch bad data vs. stale data.
High latency but low storage costs.

Data heavy workflows reduce bottlenecks and increase throughput:
Read replicas (kept uptodate with replication logs) for increased throughput and we can use data sharding and/or partitioning for.

CHALLENGE:

- Schema variation and drift. Use a registry, incremental parsing and small probe samples to detect changes early.
- Standards for data validation + cleaning - outliers, bad / conflicting data, unlikely or missing values
- Expose date last updated shows how fresh data is (latency has cost implications)

STEPS:
1) Raw documents sit in an s3 bucket - source of truth.
2) Normalised JSON (structured data warehouse (Snowflake/BigQuery/Redshift) or SQL pre-aggregated tables) for analytics and reporting.
    - **Analytical / slice-and-dice:** “How do prices vary by geography, provider, or over time?” → use an aggregated store for slicing (or OLAP-style cubes keybed) by provider, geography or time.
3) Indexing service loads data into Elasticsearch (or OpenSearch) - which is great at short-term storage, massive datasets and filtering / searching.
    - **Operational lookups:** What will *this* procedure cost for *this* member at *this* provider?” → hits search index + cache.To reduce compute and latency

REDUCE QUERY LOAD ON ELASTICSEARCH AND THE WAREHOUSE

- Pre-compute common queries, hot paths and caching them aggressively
- Use Redis or similar for application-level caching to reduce load on the search
- CDN for public/unauthenticated or semi-static content

API integrations

Use OAuth / OIDC for authentication. After login, the client receives a short-lived **JWT access token** and a longer-lived **refresh token**. API gateway or edge service validates the JWT on each request.

Expose a REST or GraphQL API for internal / external counterparties
Support webhooks or event notifications so customers can react to changes (e.g., “new rates available”, “contract updated”).
Integrate with external SDKs/APIs

Microservice boundary examples

- **AdminService** – user/tenant management, roles/permissions, feature flags.
- **SearchService** – search and lookup of prices/providers.
- **EstimateService** – applies insurance logic on top of price data.
- **LookupService** – reference data: codes, providers, locations, plans.

Example of SearchService:
Customer Informs us of what insurance plan, provider, what service type they are after
Query hits API with elastic search (dependency):

```
GET /search/procedures?q=mri&location=DC&insurance_plan_id=plan_123
  &service_date=2025-11-20&page=1&page_size=20
Authorization: Bearer <JWT>
```

Hits the **SearchService** API which retrieve rates from cache warehouse (check Redis) or ElasticSearch where we query an index (table) or **warehouse / SQL** (for exact rate lookups).

ElasticSearch potentially returns minimum medium price (or timeout, errors, circuit breaker which fail gracefully instead of taking down the endpoint)
But it is not a source of truth, it's a read optimised projection of search.

Estimate service
Calls the SearchService as if it’s a dependency (it doesn't talk to ElasticSearch)
Applies insurance deductibles (do not cache these as dynamic) cache **underlying negotiated rates** instead.

Returns a normalized price payload in JSON to the client and a confidence score (score goes down if service is down).

For quite < 300–500 ms p95 for `/estimate` results - cached lookup when possible. Avoiding heavy aggregations Efficient / very selective elastic search querying (exact filters with small result set)

Infrastructure

The services could sit inside a docker container.
Kubernetes container orchestration is good for serverless horizontal scaling / rolling updates / resource isolation.

Frontend

Role based access: How does the experience differ for enterprise clients?

- A consumer can search costs, providers - based on different insurance plans + geography - keep the design KISS.
- Enterprise - dashboards, analytics, contract management, bulk workflow + exporting options

RBAC determines which components hydrate, which APIs fire, and data visibility rules.

Performance and Page speed:

1. Hydrate non-critical widgets below the fold
2. Asynchronous data load or background prefetching for frequently-hit queries
3. CDN for public/unauthenticated or semi-static content
4. Client-side caching for recently-viewed providers or procedures.

Accessibility WCAG and A to Y
Guided flows that simplify medical vocab + jargon into human-readable language.
Mobile responsiveness and app

Compliance and Privacy

Monitoring of ingest failures and data in transit, quality metrics
Throttling of api usage
Audit trail
Resiliency and backups
Scalability
Design for failover - rollback to stable snapshot
Monitoring / alerts grifana