GENERATIVE FORECAST EXPLANATION SYSTEM OPTIMISATION

Benchmark Results Summary (Example Explanation)

Step	Time (seconds)	Status
Data Preprocessing	~0.01–0.03	Success
EDA	~0.6–1.5	Success
Prophet Forecasting	~1–3	Success
Actual vs Forecast Merge	~0.005	Success
WHY Chatbot Reasoning	~0.001–0.02	Success

Summary:

- EDA and Forecasting are the heaviest steps.
- Merging, Chatbot, and Preprocessing are lightweight.
- Prophet model fitting is usually the slowest component.
- Benchmarking helped identify which tasks needed performance improvement.

Benchmarking revealed that forecasting and EDA consumed the most time, while preprocessing and chatbot steps were efficient.

Optimization Steps I Implemented

---

1) Data Preprocessing Optimization

What I improved:

- The CSV file is loaded only once instead of multiple times.
- Used pd.to_datetime() for fast and efficient date conversion.
- Prevented preprocessing from repeating inside other sections like EDA and Forecasting tabs.

Benefit:
Faster initial loading and reduced repeated work.

What I changed:

- I moved all data loading and preprocessing inside a single block at the top:
- df = pd.read_csv(uploaded)
- df["ds"] = pd.to_datetime(df["ds"])
- I stopped repeating this inside the EDA and Forecasting tabs.
- I ensured the dataframe (df) is reused across the app.

How this optimizes performance:

- Previously Streamlit re-ran preprocessing every time a tab switched.
- By preprocessing only once, the app became faster and avoided redundant work.

---

2) EDA Optimization

What I improved:

- Used vectorized Z-score calculation, which is faster than using loops.

```
df["zscore"] = (df["y"] - df["y"].mean()) / df["y"].std()
```

- Reused the same dataframe for all EDA operations and visualizations.
- Calculated correlations only once to avoid unnecessary heavy computations.

Benefit:
Lower CPU usage and faster display of charts and statistics.

- I stored the dataframe once and reused it for:
    - summary statistics
    - distribution plots

    - line charts
    - outlier detection
    - correlation matrix

How this optimizes performance:

- Vectorized operations run in milliseconds instead of loops.
- Reusing the same dataframe removes recalculations.
- Fewer heavy operations = faster EDA rendering.

---

3) Prophet Forecasting Optimization

What I improved:

- Enabled only relevant seasonal patterns: weekly and yearly.
- Removed daily seasonality, which slows down the model without adding value.
- Trained the Prophet model only once per session rather than multiple times.

What I changed:

- I limited Prophet to only weekly and yearly seasonality:

- Prophet(yearly_seasonality=True, weekly_seasonality=True)
- I removed daily seasonality (which slows the model).
- I trained the model only once per run:

  m.fit(df)

How this optimizes performance:

- Fewer seasonality components = fewer parameters to learn.
- Prophet runs faster and uses less CPU.
- Forecasts generate more quickly.

Benefit:
Faster model training and quicker generation of forecasts.

---

4) Merge Optimization

What I improved:

- Combined forecast data and actual data once and reused it across the application.
- Merged only the required columns to keep memory usage low.

What I changed:

- I merged actual and forecasted data only once:
- merged = pd.merge(forecast, df, on="ds", how="left")
- I stored merged for reuse by the chatbot.

How this optimizes performance:

- Without this, the merge would run repeatedly every time the chatbot responded.
- Now the chatbot uses premerged data instantly.

Benefit:
Chatbot and analysis steps access forecast data instantly without recomputing anything.

---

5) Chatbot Optimization

What I improved:

- Used a lightweight rule-based reasoning system.
- Did not use any heavy machine learning operations inside the chatbot.
- Avoided re-running the forecasting model during each chatbot query.

What I changed:

- I wrote a lightweight rule-based explanation function.

- I used simple comparisons:
- if row["weekly"] > 0:
- reasoning.append("Weekly pattern increased demand.")
- I avoided running Prophet again inside chatbot queries.

How this optimizes performance:

- The chatbot only reads data — it does not calculate anything heavy.
- Responses are instant because no forecasting is redone.

Benefit:
Chatbot replies instantly and smoothly.

---

6) UI Optimization

What I improved:

- Limited DataFrame previews using .head() to avoid loading hundreds of rows at once.
- Used short captions instead of additional charts to reduce rendering time.
- Avoided unnecessary heavy UI components.

What I changed:

- Limited heavy DataFrame rendering:
- st.dataframe(df.head())
- Used minimal charts and added short captions instead.
- Avoided repeating large table or image renderings.

How this optimizes performance:

- Streamlit UI loads faster.
- Less memory usage.
- Smooth navigation between tabs.

Benefit:
The interface loads faster and switching between tabs is smoother.

Before Optimization

- EDA and Forecasting were slower.
- Some calculations repeated unnecessarily.
- Prophet model took longer to compute.
- Chatbot might recompute values.
- UI felt slightly slower.

After Optimization

- Execution time reduced significantly.
- Data processed only once.

- Forecast model runs smoother.
- Chatbot explanations are instant.
- Full app feels faster and more responsive.