class OrchestratorAgent:

    def \_\_init\_\_(self, gsheet\_url: str, semantic\_layer\_path: str, api\_key: str):

        self.query\_agent = QueryUnderstandingAgent(gsheet\_url, semantic\_layer\_path)

        self.data\_agent = DataRetrievalAgent(gsheet\_url)

        self.analysis\_agent = AnalysisAgent(load\_semantic\_layer(semantic\_layer\_path))

        self.semantic\_layer = load\_semantic\_layer(semantic\_layer\_path)

        self.narrator\_llm = ChatOpenAI(model="gpt-4o-mini", temperature=0.2)

        # Configure intent handlers

        self.intent\_handlers = {

            IntentType.FETCH\_METRIC.value: self.\_handle\_fetch\_metric,

            IntentType.COMPARE\_METRIC.value: self.\_handle\_compare\_metric,

            IntentType.RANK\_ENTITIES.value: self.\_handle\_rank\_entities,

            IntentType.THRESHOLD\_CHECK.value: self.\_handle\_threshold\_check,

            IntentType.LIST\_ENTITIES\_BY\_CRITERIA.value: self.\_handle\_list\_entities,

            IntentType.TREND\_ANALYSIS.value: self.\_handle\_trend\_analysis,

            IntentType.SUMMARIZE\_METRIC.value: self.\_handle\_summarize,

            IntentType.VISUALIZE\_METRIC.value: self.\_handle\_visualize,

            IntentType.PREDICT\_METRIC.value: self.\_handle\_predict,

            IntentType.DIAGNOSE\_METRIC.value: self.\_handle\_diagnose,

        }

    def process(self, state: AgentState) -> AgentState:

        try:

            # Step 1: Query Understanding

            state.extracted\_context = self.query\_agent.process(state.user\_query)

            logging.info(f"Extracted context: {state.extracted\_context}")

            # Step 2: Data Retrieval

            state.data = self.data\_agent.fetch\_data(state.extracted\_context)

            logging.info(f"Retrieved {len(state.data)} records")

            self.\_working\_df = state.data

            self.\_raw\_df = getattr(self.data\_agent, "\_last\_full\_df", None)

            # Step 3: Analysis

            all\_metrics = [m for ctx in state.extracted\_context['query\_analysis'] for m in ctx.get('metrics', [])]

            if not all\_metrics:

                all\_metrics = ["efficiency"]

            metric\_results = self.analysis\_agent.compute\_metrics(

                state.data,

                metrics=all\_metrics,

                context=state.extracted\_context

            )

            state.analysis\_results = {}

            logging.info(f"Analysis results: {metric\_results}")

            # Step 4: Intent-Specific Processing

            for intent\_ctx in state.extracted\_context["query\_analysis"]:

                intent = intent\_ctx["intent"]

                intent\_data = metric\_results.get(intent, pd.DataFrame())

                if intent in self.intent\_handlers:

                    handler\_result = self.intent\_handlers[intent](intent\_data, intent\_ctx)

                    state.analysis\_results[intent] = handler\_result

                else:

                    state.analysis\_results[intent] = self.\_default\_handler(intent\_data, intent\_ctx)

            # Step 5: Generate Final Report

            state.report = self.\_generate\_conversational\_report(state)

            return state

        except Exception as e:

            state.errors.append(f"Processing error: {str(e)}")

            logging.error(f"Processing failed: {str(e)}")

            return state

    # --- Intent Handlers ---

    def \_handle\_fetch\_metric(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No data found for specified criteria"}

        return {

            "type": "metric\_value",

            "data": data.to\_dict(orient='records'),

            "support\_numbers": data.iloc[0].get("support\_numbers", "") if not data.empty else ""

        }

    def \_handle\_compare\_metric(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No data available for comparison"}

        try:

            ents = [e for e in context.get("entities", []) if e.get("value") not in (None, "all")]

            raw = (context.get("user\_query") or "")

            tokens = re.findall(r"\b[A-Z]{2}\b", raw)

            for t in tokens:

                if all(str(e["value"]).upper() != t for e in ents):

                    ents.append({"field": "country\_code", "value": t})

            wanted = {str(e["value"]).upper() for e in ents}

            if len(wanted) < 2:

                return {"error": "Need at least two entities to compare"}

            records = data.to\_dict("records")

            compare\_rows = [r for r in records if str(r.get("field\_value")).upper() in wanted]

            if len(compare\_rows) < 2:

                return {"error": "Could not find data for all requested entities"}

            return {

                "type": "comparison",

                "data": compare\_rows,

                "metadata": {

                    "entities": [{"field": "country\_code", "value": v} for v in sorted(wanted)],

                    "metric": data.iloc[0]["metric"] if not data.empty else "unknown"

                }

            }

        except Exception as e:

            return {

                "error": f"Comparison failed: {str(e)}",

                "original\_data": data.to\_dict(orient='records') if not data.empty else []

            }

    def \_handle\_rank\_entities(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No data available for ranking"}

        query = context.get("user\_query", "").lower()

        is\_reverse = False

        if any(word in query for word in ["top", "high", "highest"]):

            is\_reverse = True

        n = self.\_extract\_ranking\_n(query)

        try:

            clean\_data = data.dropna(subset=['value'])

            sorted\_data = clean\_data.sort\_values('value', ascending=not is\_reverse)

            ranked = sorted\_data.head(n)

            return {

                "type": "ranking",

                "data": ranked.to\_dict(orient='records'),

                "metadata": {

                    "sort\_order": "descending" if is\_reverse else "ascending",

                    "count": len(ranked)

                }

            }

        except Exception as e:

            return {

                "error": f"Ranking failed: {str(e)}",

                "original\_data": data.to\_dict(orient='records') if not data.empty else []

            }

    def \_handle\_threshold\_check(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No data available for threshold check"}

        try:

            query = context.get("user\_query", "").lower()

            threshold = self.\_extract\_threshold(query)

            above\_threshold = data[data['value'] > threshold]

            below\_threshold = data[data['value'] <= threshold]

            return {

                "type": "threshold\_check",

                "data": data.to\_dict(orient='records'),

                "metadata": {

                    "threshold": threshold,

                    "count\_above": len(above\_threshold),

                    "count\_below": len(below\_threshold),

                    "count\_total": len(data)

                }

            }

        except Exception as e:

            return {

                "error": f"Threshold check failed: {str(e)}",

                "original\_data": data.to\_dict(orient='records') if not data.empty else []

            }

    def \_handle\_list\_entities(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No entities match the criteria"}

        return {

            "type": "entity\_list",

            "data": data.to\_dict(orient='records'),

            "metadata": {

                "count": len(data),

                "criteria": context.get("user\_query", "")

            }

        }

    def \_handle\_trend\_analysis(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No data available for trend analysis"}

        try:

            sorted\_data = data.sort\_values('field\_value')

            values = sorted\_data['value'].tolist()

            trend = "increasing" if values[-1] > values[0] else "decreasing" if values[-1] < values[0] else "stable"

            return {

                "type": "trend\_analysis",

                "data": data.to\_dict(orient='records'),

                "metadata": {

                    "trend": trend,

                    "time\_period": context.get("timeframes", [{}])[0].get("period\_in\_query", "unknown")

                }

            }

        except Exception as e:

            return {

                "error": f"Trend analysis failed: {str(e)}",

                "original\_data": data.to\_dict(orient='records') if not data.empty else []

            }

    def \_handle\_summarize(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No data available to summarize"}

        try:

            avg = data['value'].mean()

            count = len(data)

            std\_dev = data['value'].std()

            return {

                "type": "summary",

                "data": data.to\_dict(orient='records'),

                "metadata": {

                    "average": avg,

                    "count": count,

                    "standard\_deviation": std\_dev,

                    "metric": data.iloc[0]["metric"] if not data.empty else "unknown"

                }

            }

        except Exception as e:

            return {

                "error": f"Summary failed: {str(e)}",

                "original\_data": data.to\_dict(orient='records') if not data.empty else []

            }

    def \_handle\_visualize(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No data available for visualization"}

        try:

            viz\_data = []

            for \_, row in data.iterrows():

                viz\_data.append({

                    "x": row["field\_value"],

                    "y": row["value"],

                    "label": f"{row['field\_value']}: {row['value']:.2f}",

                    "support": row.get("support\_numbers", "")

                })

            return {

                "type": "visualization",

                "data": viz\_data,

                "metadata": {

                    "chart\_type": self.\_determine\_chart\_type(context),

                    "metric": data.iloc[0]["metric"] if not data.empty else "unknown"

                }

            }

        except Exception as e:

            return {

                "error": f"Visualization failed: {str(e)}",

                "original\_data": data.to\_dict(orient='records') if not data.empty else []

            }

    def \_handle\_predict(self, data: pd.DataFrame, context: Dict) -> Dict:

        try:

            entity\_field, entity\_value = self.\_get\_primary\_entity(context, default\_field="country\_code")

            if not entity\_value:

                return {"error": "No entity provided (e.g., country code). Please specify like 'BR'."}

            ts = self.\_monthly\_efficiency\_ts(entity\_field, entity\_value)

            if ts.empty or len(ts.dropna()) < 3:

                return {"error": f"Not enough history to forecast for {entity\_field}={entity\_value}."}

            try:

                model = ARIMA(ts, order=(1, 1, 1), enforce\_stationarity=False, enforce\_invertibility=False)

                fitted = model.fit()

            except Exception as e:

                return {"error": f"ARIMA fitting failed: {str(e)}"}

            n\_steps = 2

            fcast = fitted.get\_forecast(steps=n\_steps)

            f\_vals = pd.Series(fcast.predicted\_mean, index=pd.date\_range(

                start=ts.index[-1] + pd.offsets.MonthBegin(1),

                periods=n\_steps, freq="MS"

            ))

            f\_vals = f\_vals.clip(lower=0.0, upper=1.0)

            hist\_tail = ts.tail(6).copy()

            hist\_df = pd.DataFrame({

                "month": [d.strftime("%b %Y") for d in hist\_tail.index],

                "efficiency": (hist\_tail \* 100).round(2).tolist()

            })

            f\_df = pd.DataFrame({

                "month": [d.strftime("%b %Y") for d in f\_vals.index],

                "forecast\_efficiency": (f\_vals \* 100).round(2).tolist()

            })

            prompt = f"""

            You are generating a short, precise business trend explanation for vehicle efficiency.

            Context:

            - Entity: {entity\_field} = {entity\_value}

            - Historical monthly efficiency (last {len(hist\_df)} rows, in %):

            {hist\_df.to\_string(index=False)}

            - Forecast for next {len(f\_df)} months (in %, numeric and clipped to [0, 100]):

            {f\_df.to\_string(index=False)}

            Instructions:

            1) In 3–5 crisp sentences, summarize the direction (rising/falling/stable), magnitude (approximate pp change),

              and any month-over-month nuance visible.

            2) Do NOT invent extra numbers. Only refer to the values above.

            3) End with a one-sentence implication.

            """

            llm\_resp = self.narrator\_llm.invoke(prompt)

            narrative = (llm\_resp.content or "").strip()

            result\_records = []

            for m, v in f\_vals.items():

                result\_records.append({

                    "field": entity\_field,

                    "field\_value": entity\_value,

                    "metric": "efficiency",

                    "period": m.strftime("%Y-%m"),

                    "value": float(v),

                    "pretty\_value": f"{v\*100:.2f}%",

                })

            return {

                "type": "prediction",

                "entity": {"field": entity\_field, "value": entity\_value},

                "historical": hist\_df.to\_dict(orient="records"),

                "forecast": result\_records,

                "metadata": {

                    "model": "ARIMA(1,1,1)",

                    "horizon\_months": n\_steps

                },

                "narrative": narrative,

                "data": [

                    {

                        "field": entity\_field,

                        "field\_value": f"{entity\_value} — {rec['period']}",

                        "metric": "efficiency",

                        "value": rec["value"],

                        "support\_numbers": ""

                    }

                    for rec in result\_records

                ]

            }

        except Exception as e:

            return {"error": f"Prediction failed: {str(e)}"}

    def \_handle\_diagnose(self, data: pd.DataFrame, context: Dict) -> Dict:

        if data.empty:

            return {"error": "No data available for diagnosis"}

        try:

            # ---- Pull computed ratio & supports from AnalysisAgent result ----

            row = data.iloc[0]

            metric\_label = row.get("metric", "metric")

            metric\_value = float(row.get("value", float("nan")))

            support\_str = row.get("support\_numbers", "")

            # ---- Resolve entity (e.g., 'ID');  ----

            entity\_value = None

            entity\_field = "country\_code"  # Default for country diagnosis

            for e in context.get("entities", []):

                if e.get("value") not in (None, "all"):

                    entity\_value = e["value"]

                    entity\_field = e.get("field", "country\_code")

                    break

            if not entity\_value:

                m = re.search(r"\b[A-Z]{2}\b", context.get("user\_query", ""))

                if m:

                    entity\_value = m.group(0)

            is\_percent = (

                self.\_is\_percentage\_metric(metric\_label)

                or self.\_is\_percentage\_metric(context.get("metrics", ["efficiency"])[0])

            )

            value\_str = f"{metric\_value:.2%}" if is\_percent else f"{metric\_value:,.2f}"

            # ---- Parse numerator / denominator from support string ----

            parsed = self.\_parse\_support\_numbers(support\_str) if hasattr(self, "\_parse\_support\_numbers") else {}

            num\_key, den\_key = None, None

            if "TKM" in parsed and "Total\_KM" in parsed:

                num\_key, den\_key = "TKM", "Total\_KM"

            elif len(parsed) >= 2:

                keys = list(parsed.keys())

                num\_key, den\_key = keys[0], keys[1]

            numerator = parsed.get(num\_key) if num\_key else None

            denominator = parsed.get(den\_key) if den\_key else None

            # ---- Compute peer medians (exclude the current entity) ----

            num\_med = den\_med = None

            df\_ref = getattr(self, "\_raw\_df", None).copy()

            if (

                df\_ref is not None and not df\_ref.empty and entity\_value

                and num\_key and den\_key

                and all(col in df\_ref.columns for col in [entity\_field, num\_key, den\_key, "drive\_date"])

            ):

                # Apply the SAME timeframe, but DO NOT apply entity filters

                start\_ts, end\_ts = self.\_get\_time\_window\_from\_context(context)

                df\_ref["drive\_date"] = pd.to\_datetime(df\_ref["drive\_date"])

                df\_time = df\_ref[(df\_ref["drive\_date"] >= start\_ts) & (df\_ref["drive\_date"] <= end\_ts)].copy()

                # Aggregate per entity over the timeframe

                peer\_agg = (

                    df\_time.groupby(entity\_field)

                    .agg({num\_key: "sum", den\_key: "sum"})

                    .reset\_index()

                )

                # Exclude current entity and compute medians

                peers = peer\_agg[peer\_agg[entity\_field] != entity\_value]

                if not peers.empty:

                    num\_med = float(peers[num\_key].median())

                    den\_med = float(peers[den\_key].median())

            # ---- Get trend data for recent months ----

            trend\_data = self.\_get\_recent\_trend\_data(entity\_field, entity\_value, context)

            # ---- Enhanced: Gather additional metrics for LLM analysis ----

            additional\_metrics = self.\_gather\_additional\_metrics(entity\_field, entity\_value, context)

            # ---- Enhanced: Use LLM for comprehensive analysis ----

            diagnosis\_prompt = self.\_build\_diagnosis\_prompt(

                basic\_metrics={

                    "efficiency": metric\_value \* 100,  # Convert to percentage

                    "TKM": numerator,

                    "Total\_KM": denominator,

                    "peer\_median\_TKM": num\_med,

                    "peer\_median\_Total\_KM": den\_med

                },

                additional\_metrics=additional\_metrics,

                entity\_value=entity\_value,

                trend\_data=trend\_data

            )

            # Get LLM analysis

            llm\_response = self.narrator\_llm.invoke(diagnosis\_prompt)

            narrative = llm\_response.content.strip()

            # ---- Enhanced supporting numbers with precise metrics ----

            supporting\_numbers = [

                f"Efficiency: {value\_str}",

                f"TKM: {numerator:,.0f}" if numerator else "",

                f"Total\_KM: {denominator:,.0f}" if denominator else "",

            ]

            # Add peer comparisons if available

            if num\_med:

                supporting\_numbers.append(f"Peer Median TKM: {num\_med:,.0f}")

            if den\_med:

                supporting\_numbers.append(f"Peer Median Total\_KM: {den\_med:,.0f}")

            # Add route composition percentages

            c1\_km = additional\_metrics.get('C1\_kms\_Collected', 0)

            c2\_km = additional\_metrics.get('C2\_kms\_Collected', 0)

            c3\_km = additional\_metrics.get('C3\_kms\_Collected', 0)

            total\_km\_context = c1\_km + c2\_km + c3\_km

            if total\_km\_context > 0:

                c1\_pct = (c1\_km / total\_km\_context) \* 100

                c2\_pct = (c2\_km / total\_km\_context) \* 100

                c3\_pct = (c3\_km / total\_km\_context) \* 100

                supporting\_numbers.append(f"Route Composition: C1: {c1\_pct:.1f}% / C2: {c2\_pct:.1f}% / C3: {c3\_pct:.1f}%")

            # Add driver performance metrics

            if 'driver\_performance\_spread' in additional\_metrics:

                supporting\_numbers.append(f"Driver Performance Spread: {additional\_metrics['driver\_performance\_spread']:.1f}pp")

                if 'top\_driver\_efficiency' in additional\_metrics:

                    supporting\_numbers.append(f"Top Driver Efficiency: {additional\_metrics['top\_driver\_efficiency']:.1f}%")

                if 'bottom\_driver\_efficiency' in additional\_metrics:

                    supporting\_numbers.append(f"Bottom Driver Efficiency: {additional\_metrics['bottom\_driver\_efficiency']:.1f}%")

            # Add SOH

            if 'SOH' in additional\_metrics:

                supporting\_numbers.append(f"SOH: {additional\_metrics['SOH']:,.0f} hours")

            # Add trend information

            if trend\_data and len(trend\_data) >= 2:

                values = [d['efficiency'] for d in trend\_data]

                months = [d['month'] for d in trend\_data]

                trend\_str = " → ".join(f"{v:.1f}%" for v in values)

                trend\_change = values[-1] - values[0]

                supporting\_numbers.append(f"Trend: {trend\_str} ({trend\_change:+.1f}pp change)")

            supporting\_numbers = [s for s in supporting\_numbers if s]

            return {

                "type": "diagnosis",

                "data": data.to\_dict(orient="records"),

                "metadata": {

                    "metric": metric\_label,

                    "entity": entity\_value,

                    "numerator\_key": num\_key,

                    "denominator\_key": den\_key,

                    "numerator": numerator,

                    "denominator": denominator,

                    "peer\_num\_median": num\_med,

                    "peer\_den\_median": den\_med,

                },

                "narrative": narrative,

                "supporting\_numbers": supporting\_numbers,

                "trend\_data": trend\_data

            }

        except Exception as e:

            return {

                "error": f"Diagnosis failed: {str(e)}",

                "original\_data": data.to\_dict(orient="records") if not data.empty else [],

            }

    def \_gather\_additional\_metrics(self, entity\_field: str, entity\_value: str, context: Dict) -> Dict:

        """Gather C1/C2/C3, SOH, and driver metrics from filtered dataframe"""

        additional\_metrics = {}

        try:

            df = getattr(self, "\_working\_df", None)

            if df is None or df.empty:

                return additional\_metrics

            temp = df.copy()

            if entity\_value and entity\_field in temp.columns:

                temp = temp[temp[entity\_field] == entity\_value]

            # Use the correct column names from semantic layer

            for chain\_col in ['C1\_kms\_Collected', 'C2\_kms\_Collected', 'C3\_kms\_Collected']:

                if chain\_col in temp.columns:

                    additional\_metrics[chain\_col] = temp[chain\_col].sum()

            if 'SOH' in temp.columns:

                additional\_metrics['SOH'] = temp['SOH'].sum()

            if 'user\_email' in temp.columns and 'efficiency' in temp.columns:

                driver\_stats = temp.groupby('user\_email')['efficiency'].agg(['mean', 'std']).reset\_index()

                if not driver\_stats.empty and len(driver\_stats) > 1:

                    additional\_metrics['driver\_performance\_spread'] = driver\_stats['std'].mean()

                    additional\_metrics['top\_driver\_efficiency'] = driver\_stats['mean'].max()

                    additional\_metrics['bottom\_driver\_efficiency'] = driver\_stats['mean'].min()

        except Exception as e:

            print(f"Warning: Could not gather additional metrics: {e}")

        return additional\_metrics

    def \_build\_diagnosis\_prompt(self, basic\_metrics: Dict, additional\_metrics: Dict,

                            entity\_value: str, trend\_data: List[Dict]) -> str:

          """Build prompt for precise, data-driven diagnosis with narrative structure"""

          # Calculate percentage differences for peer comparison

          tkm\_pct\_diff = self.\_calculate\_percentage\_diff(basic\_metrics.get('TKM', 0), basic\_metrics.get('peer\_median\_TKM', 0))

          total\_km\_pct\_diff = self.\_calculate\_percentage\_diff(basic\_metrics.get('Total\_KM', 0), basic\_metrics.get('peer\_median\_Total\_KM', 0))

          prompt = f"""

      Generate a data-driven operational efficiency diagnosis for {entity\_value} using EXACT numbers from the provided metrics.

      \*\*CORE METRICS:\*\*

      - Current Efficiency: {basic\_metrics.get('efficiency', 0):.2f}%

      - TKM: {basic\_metrics.get('TKM', 0):,.0f} ({tkm\_pct\_diff} below peer median)

      - Total\_KM: {basic\_metrics.get('Total\_KM', 0):,.0f} ({total\_km\_pct\_diff} below peer median)

      \*\*CONTEXT DATA:\*\*"""

          # Add topography metrics

          c1\_km = additional\_metrics.get('C1\_kms\_Collected', 0)

          c2\_km = additional\_metrics.get('C2\_kms\_Collected', 0)

          c3\_km = additional\_metrics.get('C3\_kms\_Collected', 0)

          total\_km\_context = c1\_km + c2\_km + c3\_km

          if total\_km\_context > 0:

              c1\_pct = (c1\_km / total\_km\_context) \* 100

              c2\_pct = (c2\_km / total\_km\_context) \* 100

              c3\_pct = (c3\_km / total\_km\_context) \* 100

              prompt += f"\n- Route Composition: C1: {c1\_pct:.1f}% / C2: {c2\_pct:.1f}% / C3: {c3\_pct:.1f}%"

          # Add driver metrics

          if 'driver\_performance\_spread' in additional\_metrics:

              prompt += f"\n- Driver Performance Spread: {additional\_metrics['driver\_performance\_spread']:.1f}pp"

              if 'top\_driver\_efficiency' in additional\_metrics and 'bottom\_driver\_efficiency' in additional\_metrics:

                  prompt += f" (Top: {additional\_metrics['top\_driver\_efficiency']:.1f}%, Bottom: {additional\_metrics['bottom\_driver\_efficiency']:.1f}%)"

          # Add SOH

          if 'SOH' in additional\_metrics:

              soh = additional\_metrics['SOH']

              prompt += f"\n- SOH: {soh:,.0f} hours"

          # Add trend

          if trend\_data and len(trend\_data) >= 2:

              values = [d['efficiency'] for d in trend\_data]

              months = [d['month'] for d in trend\_data]

              trend\_str = " → ".join(f"{v:.1f}%" for v in values)

              trend\_change = values[-1] - values[0]

              prompt += f"\n- Trend: {trend\_str} ({trend\_change:+.1f}pp over {len(values)} months)"

          prompt += f"""

      \*\*CRITICAL INSTRUCTIONS:\*\*

      1. Use ONLY the exact numbers provided above - no estimations or assumptions

      2. ABSOLUTELY NO vague language: no "may", "could", "suggest", "potential", "might"

      3. Structure the analysis in this exact format:

      \*\*Current Standing:\*\*

      [Entity]'s operational efficiency stands at [X]%, placing it [Y] percentage points below the regional average of [Z]%.

      \*\*Recent Performance:\*\*

      Efficiency trends show [Month1]: [A]% → [Month2]: [B]% → [Month3]: [C]%, representing a [±X] percentage point change over the period.

      \*\*Root Cause Analysis:\*\*

      - \*\*Operational Effectiveness:\*\* [Driver spread: Xpp] combined with [SOH utilization context] contributes [Y] percentage points to the efficiency gap

      - \*\*Route Composition:\*\* [C1: A%] / [C2: B%] / [C3: C%] distribution creates [Z] percentage point efficiency impact

      - \*\*Scale Deficiency:\*\* TKM at [Number] ([X]% below peer median) and Total\_KM at [Number] ([Y]% below median) account for [Z] percentage points of the performance gap

      \*\*Performance Decomposition:\*\*

      - Operational factors: [X]pp impact ([Y]% of total gap)

      - Route mix: [X]pp impact ([Y]% of total gap)

      - Market scale: [X]pp impact ([Y]% of total gap)

      \*\*Conclusion:\*\*

      [Primary factor] presents the primary opportunity, addressing [X] percentage points of the efficiency gap. [Secondary factor] offers additional potential for [Y] points. Based on peer performance benchmarks, achieving [Z]% efficiency represents a realistic target.

      4. Use only factual, quantitative statements based on the provided numbers

      5. For peer comparisons, use relative percentage point differences, not vague descriptions

      6. All impacts must be quantified in percentage points with exact numbers

      """

          return prompt

    def \_calculate\_percentage\_diff(self, value: float, median: float) -> str:

        """Calculate percentage difference for prompt"""

        if median == 0 or value is None:

            return "N/A"

        pct\_diff = ((median - value) / median) \* 100

        direction = "above" if pct\_diff < 0 else "below"

        return f"{abs(pct\_diff):.0f}% {direction}"

    def \_get\_recent\_trend\_data(self, entity\_field: str, entity\_value: str, context: Dict) -> List[Dict]:

        """Get efficiency trend data for the last 3 months"""

        try:

            df = getattr(self, "\_working\_df", None)

            if df is None or df.empty or "drive\_month" not in df.columns:

                return []

            temp = df.copy()

            if entity\_value and entity\_field in temp.columns:

                temp = temp[temp[entity\_field] == entity\_value]

            # Ensure drive\_month is properly formatted

            temp["drive\_month"] = pd.to\_datetime(temp["drive\_month"])

            recent\_months = temp["drive\_month"].dropna().unique()

            recent\_months.sort()

            recent\_months = recent\_months[-3:]  # Get last 3 months

            trend\_data = []

            for month in recent\_months:

                month\_data = temp[temp["drive\_month"] == month]

                if not month\_data.empty:

                    tkm\_sum = month\_data["TKM"].sum()

                    total\_km\_sum = month\_data["Total\_KM"].sum()

                    efficiency = (tkm\_sum / total\_km\_sum) \* 100 if total\_km\_sum != 0 else 0

                    trend\_data.append({

                        "month": month.strftime("%b %Y"),

                        "efficiency": efficiency,

                        "TKM": tkm\_sum,

                        "Total\_KM": total\_km\_sum

                    })

            return trend\_data

        except Exception as e:

            print(f"Error getting trend data: {e}")

            return []

    def \_default\_handler(self, data: pd.DataFrame, context: Dict) -> Dict:

        return {

            "type": "generic\_results",

            "data": data.to\_dict(orient='records') if not data.empty else [],

            "support\_numbers": data.iloc[0].get("support\_numbers", "") if not data.empty else ""

        }

    # --- Helper Methods ---

    def \_get\_primary\_entity(self, context: Dict, default\_field: str = "country\_code") -> tuple:

        entity\_field = default\_field

        entity\_value = None

        try:

            if isinstance(context, dict) and "query\_analysis" in context:

                q = context["query\_analysis"][0] if context.get("query\_analysis") else {}

            else:

                q = context or {}

            entities = [

                e for e in q.get("entities", [])

                if isinstance(e, dict)

                and e.get("value") not in (None, "all")

                and e.get("condition\_type") != "numeric\_filter"

            ]

            if entities:

                entity\_field = entities[0].get("field", default\_field)

                entity\_value = entities[0].get("value")

            if not entity\_value:

                m = re.search(r"\b[A-Z]{2}\b", q.get("user\_query", ""))

                if m:

                    entity\_field = default\_field

                    entity\_value = m.group(0)

        except Exception as e:

            logging.error(f"\_get\_primary\_entity error: {e}")

        return entity\_field, entity\_value

    def \_monthly\_efficiency\_ts(self, entity\_field: str, entity\_value: str) -> pd.Series:

        df = getattr(self, "\_working\_df", None)

        if df is None or df.empty:

            return pd.Series(dtype=float)

        temp = df.copy()

        if entity\_field in temp.columns and entity\_value:

            temp = temp[temp[entity\_field] == entity\_value]

        required\_cols = {"drive\_month", "TKM", "Total\_KM"}

        if not required\_cols.issubset(set(temp.columns)):

            return pd.Series(dtype=float)

        ts = (

            temp.dropna(subset=["drive\_month"])

                .groupby("drive\_month", as\_index=True)

                .agg(TKM=("TKM", "sum"), Total\_KM=("Total\_KM", "sum"))

        )

        if ts.empty:

            return pd.Series(dtype=float)

        ts["efficiency"] = ts["TKM"] / ts["Total\_KM"]

        ts = ts["efficiency"].dropna().sort\_index()

        return ts

    def \_get\_time\_window\_from\_context(self, context: Dict) -> tuple:

        try:

            q = context.get("query\_analysis", [])[0]

            tfs = q.get("timeframes", [])

            if tfs:

                start = pd.to\_datetime(tfs[0].get("start"))

                end = pd.to\_datetime(tfs[0].get("end"))

                return start, end

        except Exception:

            pass

        return pd.to\_datetime("1970-01-01"), pd.Timestamp.now().tz\_localize(None)

    def \_parse\_support\_numbers(self, s: str) -> dict:

        if not s:

            return {}

        pairs = re.findall(r'([A-Za-z\_]+)\s\*:\s\*([\-]?\d+(?:\.\d+)?)', s)

        return {k: float(v) for k, v in pairs}

    def \_is\_percentage\_metric(self, metric\_name: str) -> bool:

        pct\_metrics = {"efficiency", "c1\_km\_perc"}

        return metric\_name.lower() in pct\_metrics

    def \_extract\_ranking\_n(self, query: str) -> int:

        q = (query or "").lower()

        m = re.search(r'(?:top|bottom|lowest|highest)\s+(\d+)|(\d+)\s+(?:lowest|highest)', q)

        if m:

            return int(m.group(1) or m.group(2))

        if any(w in q for w in ["highest", "lowest"]):

            return 1

        return 5

    def \_extract\_threshold(self, query: str) -> float:

        match = re.search(

            r'(?:above|below|>|<|>=|<=|==)\s\*(\d+(?:\.\d+)?)',

            query.lower()

        )

        return float(match.group(1)) if match else 0.0

    def \_determine\_chart\_type(self, context: Dict) -> str:

        intent = context.get("intent", "")

        timeframes = context.get("timeframes", [])

        if intent == IntentType.TREND\_ANALYSIS.value and timeframes:

            return "line\_chart"

        elif intent == IntentType.COMPARE\_METRIC.value:

            return "bar\_chart"

        elif intent == IntentType.RANK\_ENTITIES.value:

            return "horizontal\_bar\_chart"

        else:

            return "bar\_chart"

    def \_generate\_conversational\_report(self, state: AgentState) -> str:

        if not state.analysis\_results:

            return "No results found for your query."

        qa\_list = (state.extracted\_context or {}).get("query\_analysis") or []

        if not qa\_list:

            return "No query analysis available."

        query\_ctx = qa\_list[0]

        intent = query\_ctx["intent"]

        primary\_result = None

        if IntentType.DIAGNOSE\_METRIC.value in state.analysis\_results:

            primary\_result = state.analysis\_results[IntentType.DIAGNOSE\_METRIC.value]

        else:

            for k in state.analysis\_results:

                primary\_result = state.analysis\_results[k]

                break

        if not primary\_result or "error" in primary\_result:

            return primary\_result.get("error", "No results found for your query.")

        report\_parts = []

        if intent == IntentType.DIAGNOSE\_METRIC.value:

            if "narrative" in primary\_result:

                report\_parts.append(primary\_result["narrative"])

            if primary\_result.get("supporting\_numbers"):

                report\_parts.append("\n\*\*Supporting Numbers:\*\*")

                for num in primary\_result["supporting\_numbers"]:

                    report\_parts.append(f"- {num}")

        else:

            results\_data = primary\_result.get("data", [])

            if results\_data:

                response = self.\_generate\_conversational\_response(query\_ctx, results\_data)

                report\_parts.append(response)

                table = self.\_generate\_results\_table(query\_ctx, results\_data)

                if table:

                    report\_parts.append("\n" + table)

        suggestions = self.\_generate\_suggested\_queries(state.extracted\_context)[:3]

        if suggestions:

            report\_parts.append("\n\*\*Suggested follow-up queries:\*\*")

            report\_parts.extend([f"{i}. {q}" for i, q in enumerate(suggestions, 1)])

        return "\n".join(report\_parts)

    def \_generate\_conversational\_response(self, query\_ctx: Dict, results\_data: List[Dict]) -> str:

        intent = query\_ctx["intent"]

        metric\_name = self.\_get\_friendly\_name(query\_ctx["metrics"][0])

        if intent == IntentType.RANK\_ENTITIES.value:

            rows = [r for r in results\_data if pd.notnull(r.get("value"))]

            if not rows:

                return "No results found."

            q = (query\_ctx.get("user\_query") or "").lower()

            want\_highest = any(w in q for w in ["top", "high", "highest"])

            best = max(rows, key=lambda r: r["value"]) if want\_highest else min(rows, key=lambda r: r["value"])

            metric\_name = self.\_get\_friendly\_name(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else "Metric"

            ent\_label   = self.\_get\_friendly\_name(best.get("field", "entity"))

            ent\_value   = str(best.get("field\_value"))

            is\_percent  = self.\_is\_percentage\_metric(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else True

            fmt         = (lambda v: f"{v:.2%}") if is\_percent else (lambda v: f"{v:,.2f}")

            if want\_highest:

                return f"{ent\_value} has the highest {metric\_name} at {fmt(best['value'])}."

            else:

                return f"{ent\_value} has the lowest {metric\_name} at {fmt(best['value'])}."

        if intent == IntentType.TREND\_ANALYSIS.value:

            try:

                return self.\_format\_mom\_trend(query\_ctx, results\_data)

            except Exception as e:

                print(f"Error in \_format\_mom\_trend: {e}")

        if intent == IntentType.COMPARE\_METRIC.value:

            rows = [r for r in results\_data if pd.notnull(r.get("value"))]

            if len(rows) >= 2:

                rows = sorted(rows, key=lambda x: x["value"], reverse=True)

                metric\_name = self.\_get\_friendly\_name(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else "Metric"

                is\_percent = self.\_is\_percentage\_metric(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else True

                fmt = (lambda v: f"{v:.2%}") if is\_percent else (lambda v: f"{v:,.2f}")

                winner = rows[0]

                winner\_name = str(winner["field\_value"])

                winner\_val  = fmt(winner["value"])

                names\_in\_results = [str(r["field\_value"]) for r in rows]

                if len(rows) == 2:

                    loser = rows[1]

                    return f"{winner\_name} has higher {metric\_name} ({winner\_val}) than {loser['field\_value']} ({fmt(loser['value'])})."

                else:

                    among = ", ".join(names\_in\_results)

                    return f"{winner\_name} has the highest {metric\_name} at {winner\_val} among {among}."

        entities = query\_ctx.get("entities", [])

        metric\_value = results\_data[0].get("value", 0)

        is\_percent = self.\_is\_percentage\_metric(query\_ctx["metrics"][0])

        value\_str = f"{metric\_value:.2%}" if is\_percent else f"{metric\_value:,.2f}"

        if entities and entities[0]["value"] != "all":

            entity\_value = entities[0]["value"]

            return f"The {metric\_name} for {entity\_value} is {value\_str}"

        else:

            return f"The {metric\_name} is {value\_str}"

    def \_generate\_results\_table(self, query\_ctx: Dict, results\_data: List[Dict]) -> str:

        if not results\_data:

            return ""

        entity\_field = query\_ctx.get("entities", [{}])[0].get("field", "entity")

        entity\_name  = self.\_get\_friendly\_name(entity\_field)

        if query\_ctx.get("intent") == IntentType.TREND\_ANALYSIS.value:

            entity\_name = "Month"

            try:

                tmp = pd.DataFrame(results\_data).copy()

                tmp["\_m"] = pd.to\_datetime(tmp["field\_value"].ast)

                tmp["\_m"] = pd.to\_datetime(tmp["field\_value"].astype(str).str[:7], errors="coerce")

                results\_data = tmp.sort\_values("\_m").drop(columns=["\_m"]).to\_dict(orient="records")

            except Exception:

                pass

        distinct\_vals = {str(item.get("field\_value")) for item in results\_data if "field\_value" in item}

        multi\_entity  = (query\_ctx.get("intent") == IntentType.COMPARE\_METRIC.value) or (len(distinct\_vals) > 1)

        rows = []

        for item in results\_data:

            row\_entity\_value = item.get("field\_value", "N/A") if multi\_entity else (

                query\_ctx["entities"][0]["value"]

                if query\_ctx.get("entities") and query\_ctx["entities"][0]["value"] != "all"

                else item.get("field\_value", "N/A")

            )

            metric\_str = (

                f"{item.get('value', 0):.2%}"

                if self.\_is\_percentage\_metric(item.get("metric", ""))

                else f"{item.get('value', 0):.2f}"

            )

            rows.append({entity\_name: row\_entity\_value,

                        "Metric": metric\_str,

                        "Supporting Numbers": item.get("support\_numbers", "")})

        df = pd.DataFrame(rows)

        return "### Detailed Results\n" + df.to\_markdown(index=False, tablefmt="grid")

    def \_generate\_suggested\_queries(self, context: Dict) -> List[str]:

        """Generate context-aware suggested queries"""

        query\_ctx = context["query\_analysis"][0]

        intent = query\_ctx["intent"]

        metric = query\_ctx["metrics"][0]

        entities = query\_ctx.get("entities", [])

        suggestions = []

        # Base suggestions for all intents

        if entities and entities[0]["value"] != "all":

            entity\_value = entities[0]["value"]

            suggestions.extend([

                f"Show trend of {metric} for {entity\_value} over time",

                f"Compare {entity\_value} with similar entities",

                f"Breakdown components of {metric} for {entity\_value}"

            ])

        else:

            suggestions.extend([

                f"Show top 10 entities by {metric}",

                f"Compare average {metric} by region",

                f"Analyze trend of {metric} over time"

            ])

        # Intent-specific additions

        if intent == IntentType.COMPARE\_METRIC.value:

            suggestions.append("Show comparison breakdown by component")

        return suggestions[:3]  # Return only top 3 most relevant

    def \_get\_friendly\_name(self, technical\_name: str) -> str:

        """Convert technical names to friendly display names"""

        name\_map = {

            "country\_code": "Country",

            "region": "Region",

            "vehicle\_id": "Vehicle",

            "efficiency": "Efficiency",

            "TKM": "Target Kilometers",

            "Total\_KM": "Total Kilometers",

            "all": "All"

        }

        return name\_map.get(technical\_name, technical\_name.replace("\_", " ").title())

    def \_format\_mom\_trend(self, query\_ctx: Dict, results\_data: List[Dict]) -> str:

        # keep only rows with a value

        rows = [r for r in results\_data if pd.notnull(r.get("value"))]

        # sort by month-like field\_value (e.g., '2025-01', '2025-02')

        def \_parse\_month(s: str):

            try:

                # works for 'YYYY-MM' and for 'YYYY-MM-DD'; falls back gracefully

                return pd.to\_datetime(str(s)[:7], errors="coerce")

            except Exception:

                return pd.NaT

        rows = sorted(rows, key=lambda r: \_parse\_month(r.get("field\_value", "")))

        # format values as % (efficiency is percentage)

        is\_percent = self.\_is\_percentage\_metric(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else True

        fmt = (lambda v: f"{v:.2%}") if is\_percent else (lambda v: f"{v:,.2f}")

        series = " \u2192 ".join(fmt(r["value"]) for r in rows)  # \u2192 is → arrow

        # friendly header

        metric\_name = self.\_get\_friendly\_name(query\_ctx["metrics"][0]) if query\_ctx.get("metrics") else "Metric"

        entity\_value = next((e.get("value") for e in (query\_ctx.get("entities") or []) if e.get("value") not in (None, "all")), None)

        if entity\_value:

            return f"The {metric\_name} trend for {entity\_value} is {series}."

        return f"The {metric\_name} trend is {series}."