.Guildenlines to create Agent Modules.

**Query Understanding Agent**

**1. What Do the Input Elements Have?**

* **user\_query**: The exact string from the user (e.g., natural language, may include ambiguous or follow-up requests).
* **context**: Dict containing:
  + **user\_id**: Unique user identifier (from auth/session)
  + **session\_id**: Unique session/conversation ID
  + **previous\_queries**: List of last N queries (for context, disambiguation)
  + **last\_feedback**: Most recent user feedback (for personalization)
  + **timestamp**: Current query time
  + **preferences**: (Optional) User’s preferred units, formats, etc.
  + **location/role**: (Optional) User’s region, access level, etc.

**2. What Is the Input/Output Format?**

* **Input:**

{  
 "user\_query": str,  
 "context": {  
 "user\_id": str,  
 "session\_id": str,  
 "previous\_queries": list,  
 "last\_feedback": str,  
 "timestamp": str,  
 "preferences": dict,  
 "location": str,  
 "role": str  
 }  
}

* **Output:**

{  
 "intent": str, # e.g. "causal\_report"  
 "entities": dict, # e.g. {"country": "US"}  
 "metrics": list, # e.g. ["efficiency"]  
 "dimensions": list, # e.g. ["month"]  
 "filters": dict, # e.g. {"country": "US"}  
 "output\_type": str, # e.g. "report\_with\_plots\_tables"  
 "context\_used": dict, # Subset of context actually used  
 "confidence": float, # (Optional) Parsing confidence score  
 "clarification\_needed": bool # (Optional) If ambiguity detected  
}

**3. Input and Output Example**

**Input:**

{  
 "user\_query": "Can you please generate a report stating the cause of efficiency drop in the US. Please give supporting plots or tables.",  
 "context": {  
 "user\_id": "u123",  
 "session\_id": "sess456",  
 "previous\_queries": [  
 "Show me efficiency for US in 2024",  
 "Compare with 2023"  
 ],  
 "last\_feedback": "Wanted more details on weather impact",  
 "timestamp": "2025-06-26T12:41:00Z",  
 "preferences": {"output\_format": "PDF"},  
 "location": "US",  
 "role": "analyst"  
 }  
}

**Output:**

{  
 "intent": "causal\_report",  
 "entities": {"country": "US"},  
 "metrics": ["efficiency"],  
 "dimensions": ["month"],  
 "filters": {"country": "US"},  
 "output\_type": "report\_with\_plots\_tables",  
 "context\_used": {  
 "previous\_queries": [  
 "Show me efficiency for US in 2024",  
 "Compare with 2023"  
 ],  
 "last\_feedback": "Wanted more details on weather impact"  
 },  
 "confidence": 0.97,  
 "clarification\_needed": False  
}

**4. What Is the Source of Context?**

* **BigQuery (Vector Search):** Stores and retrieves user history, feedback, preferences, and session metadata.
* **Session/Auth Service:** Provides user\_id/session\_id.
* **ManagerAgent:** Orchestrates context retrieval and passes it to this agent.

**5. What Is the Technical Flow to Get Context?**

1. User submits query via UI.
2. Frontend attaches user\_id/session\_id (from session/auth).
3. ManagerAgent queries BigQuery for previous queries, feedback, preferences.
4. ManagerAgent combines all into a context dict.
5. ManagerAgent calls QueryUnderstandingAgent with user\_query and context.

**6. Summary and Best Practices**

* Use context to resolve ambiguity, personalize, and improve parsing.
* Return confidence and clarification flags for downstream error handling.
* **Integration:** ManagerAgent → QueryUnderstandingAgent (with context), returns parsed query to ManagerAgent.

**Feedback Agent**

**1. What Do the Input Elements Have?**

* **feedback**: User’s feedback (free text, rating, or structured dict).
* **query\_id**: Unique identifier for the query/report.
* **user\_id**, **session\_id**, **timestamp**: For traceability.
* **response\_metadata**: (Optional) Info about the response (e.g., report type, time taken).

**2. What Is the Input/Output Format?**

* **Input:**

{  
 "query\_id": str,  
 "user\_id": str,  
 "session\_id": str,  
 "feedback": str,  
 "timestamp": str,  
 "response\_metadata": dict  
}

* **Output:**

{  
 "status": str, # "success" or "error"  
 "message": str, # Confirmation or error message  
 "feedback\_id": str, # (Optional) Unique feedback log ID  
 "context\_updated": bool # True if context/memory was updated  
}

**3. Input and Output Example**

**Input:**

{  
 "query\_id": "xyz789",  
 "user\_id": "u123",  
 "session\_id": "sess456",  
 "feedback": "Report was helpful, but please add more weather details.",  
 "timestamp": "2025-06-26T13:00:00Z",  
 "response\_metadata": {"report\_type": "causal", "delivery\_time\_sec": 4.2}  
}

**Output:**

{  
 "status": "success",  
 "message": "Feedback recorded and context updated.",  
 "feedback\_id": "fb\_00123",  
 "context\_updated": True  
}

**4. What Is the Source of Context?**

* **BigQuery:** Feedback logs table, linked by query\_id, user\_id, session\_id.
* **ManagerAgent:** Passes feedback and metadata after user interaction.

**5. What Is the Technical Flow to Get Context?**

1. ManagerAgent sends feedback input to FeedbackAgent.
2. FeedbackAgent logs feedback to BigQuery (with all metadata).
3. Optionally, updates user/session context in BigQuery (for future personalization).
4. Returns status and feedback\_id to ManagerAgent.

**6. Summary and Best Practices**

* Always store feedback with full traceability (user, session, query).
* Use feedback to improve future responses (context update).
* **Integration:** ManagerAgent → FeedbackAgent, returns status to ManagerAgent.

**Semantic Layer & Data Retrieval Agent**

**1. What Do the Input Elements Have?**

* **query\_struct**: Dict with intent, entities, metrics, dimensions, filters (from QueryUnderstandingAgent).
* **semantic\_layer\_config**: YAML/Python dict mapping business logic (table/field names, metric definitions, join logic).
* **user/session context**: (Optional) For row-level security or personalized data access.

**2. What Is the Input/Output Format?**

* **Input:**

{  
 "metrics": list,  
 "dimensions": list,  
 "filters": dict,  
 "user\_context": dict  
}

* **Intermediate Output:**

{  
 "generated\_sql": str,  
 "semantic\_layer\_version": str,  
 "parameters": dict  
}

* **Final Output:**
  + List of dicts (tabular data), with metadata.

**3. Input and Output Example**

**Input:**

{  
 "metrics": ["efficiency"],  
 "dimensions": ["month"],  
 "filters": {"country": "US"},  
 "user\_context": {"role": "analyst"}  
}

**Intermediate Output:**

{  
 "generated\_sql": "SELECT month, efficiency FROM efficiency\_table WHERE country = 'US' ORDER BY month",  
 "semantic\_layer\_version": "v1.2",  
 "parameters": {"country": "US"}  
}

**Final Output:**

{  
 "data": [  
 {"month": "2025-01", "efficiency": 0.82},  
 {"month": "2025-02", "efficiency": 0.76},  
 {"month": "2025-03", "efficiency": 0.79}  
 ],  
 "row\_count": 3,  
 "query\_time\_sec": 0.8  
}

**4. What Is the Source of Context?**

* **Semantic Layer Config:** YAML/Python file loaded at runtime.
* **BigQuery:** Data warehouse for all queries.
* **ManagerAgent:** Passes structured query and config.

**5. What Is the Technical Flow to Get Context?**

1. ManagerAgent receives parsed query, passes to Data Retrieval Agent.
2. Data Retrieval Agent uses semantic layer to generate SQL.
3. Executes SQL on BigQuery (with user/session context if needed).
4. Returns data and intermediate artifacts (SQL, timing, row count) to ManagerAgent.

**6. Summary and Best Practices**

* Keep semantic layer versioned and testable.
* Log all generated SQL for debugging and audit.
* **Integration:** ManagerAgent → DataRetrievalAgent (with query\_struct), returns data and SQL to ManagerAgent.

**Data Analysis Agent**

**1. What Do the Input Elements Have?**

* **data**: List of dicts (tabular data) from Data Retrieval Agent.
* **analysis\_params**: (Optional) Dict specifying type of analysis (trend, causal, correlation), thresholds, or hypothesis.
* **context**: (Optional) Baseline or reference data for comparison.

**2. What Is the Input/Output Format?**

* **Input:**
  + List of dicts (tabular data), analysis\_params (dict), context (dict).
* **Intermediate Output:**
  + Arrays of computed values (e.g., trend slopes, correlation matrices, summary statistics).
* **Final Output:**

{  
 "trend": str,  
 "drop\_detected": bool,  
 "drop\_period": str,  
 "possible\_causes": list,  
 "analysis\_notes": str,  
 "intermediate\_results": dict  
}

**3. Input and Output Example**

**Input:**

{  
 "data": [  
 {"month": "2025-01", "efficiency": 0.82},  
 {"month": "2025-02", "efficiency": 0.76},  
 {"month": "2025-03", "efficiency": 0.79}  
 ],  
 "analysis\_params": {"type": "trend\_and\_causal", "threshold": 0.05}  
}

**Intermediate Output:**

{  
 "trend\_array": [-0.06, +0.03],  
 "correlations": {"weather": -0.8},  
 "mean\_efficiency": 0.79  
}

**Final Output:**

{  
 "trend": "downward",  
 "drop\_detected": True,  
 "drop\_period": "2025-02",  
 "possible\_causes": ["unusual weather"],  
 "analysis\_notes": "Significant drop in February; correlates with reported snowstorm.",  
 "intermediate\_results": {  
 "trend\_array": [-0.06, +0.03],  
 "correlations": {"weather": -0.8},  
 "mean\_efficiency": 0.79  
 }  
}

**4. What Is the Source of Context?**

* **BigQuery:** For baseline or reference data.
* **ManagerAgent:** Passes all data and analysis parameters.

**5. What Is the Technical Flow to Get Context?**

1. Receives data and params from ManagerAgent.
2. Optionally fetches baseline data from BigQuery.
3. Performs analysis (Pandas, NumPy, SciKit-Learn).
4. Returns summary and intermediate results to ManagerAgent.

**6. Summary and Best Practices**

* Return both summary and intermediate artifacts for transparency.
* Parameterize analysis for flexibility.
* **Integration:** ManagerAgent → DataAnalysisAgent (with data/params), returns analysis to ManagerAgent.

**Anomaly Detection Agent**

**1. What Do the Input Elements Have?**

* **data**: List of dicts (tabular data) from Data Retrieval or Data Analysis Agent.
* **anomaly\_params**: (Optional) Dict with thresholds, methods (z-score, IQR, ML model).
* **context**: (Optional) For reference distributions or historical norms.

**2. What Is the Input/Output Format?**

* **Input:**
  + List of dicts, anomaly\_params (dict), context (dict).
* **Intermediate Output:**
  + Z-score arrays, anomaly flags, model predictions.
* **Final Output:**

{  
 "anomalies": list,  
 "explanation": str,  
 "intermediate\_results": dict  
}

**3. Input and Output Example**

**Input:**

{  
 "data": [  
 {"month": "2025-01", "efficiency": 0.82},  
 {"month": "2025-02", "efficiency": 0.76},  
 {"month": "2025-03", "efficiency": 0.79}  
 ],  
 "anomaly\_params": {"method": "z\_score", "threshold": 2.0}  
}

**Intermediate Output:**

{  
 "z\_scores": [0.5, -2.3, 0.8],  
 "anomaly\_flags": [False, True, False]  
}

**Final Output:**

{  
 "anomalies": [  
 {"month": "2025-02", "value": 0.76, "z\_score": -2.3}  
 ],  
 "explanation": "Significant efficiency drop detected in Feb 2025.",  
 "intermediate\_results": {  
 "z\_scores": [0.5, -2.3, 0.8],  
 "anomaly\_flags": [False, True, False]  
 }  
}

**4. What Is the Source of Context?**

* **BigQuery:** For reference distributions.
* **ManagerAgent or DataAnalysisAgent:** Passes data and parameters.

**5. What Is the Technical Flow to Get Context?**

1. Receives data and params from ManagerAgent or DataAnalysisAgent.
2. Optionally fetches reference data from BigQuery.
3. Runs anomaly detection (SciKit-Learn, NumPy).
4. Returns anomalies and intermediate outputs to caller.

**6. Summary and Best Practices**

* Provide both anomaly explanations and raw flags/scores.
* Parameterize for different detection strategies.
* **Integration:** ManagerAgent or DataAnalysisAgent → AnomalyDetectionAgent, returns to caller.

**Report Generation Agent**

**1. What Do the Input Elements Have?**

* **analysis\_results**: Dict from Data Analysis Agent.
* **anomaly\_report**: Dict from Anomaly Detection Agent (optional).
* **raw\_data**: List of dicts for tables/plots.
* **report\_params**: (Optional) Dict for output format, template, user preferences.

**2. What Is the Input/Output Format?**

* **Input:**
  + Dict (analysis, anomalies, data, params).
* **Intermediate Output:**
  + Plot image files (PNG/SVG), HTML snippets, table data, narrative text.
* **Final Output:**

{  
 "report\_html": str,  
 "plots": list,  
 "tables": list,  
 "summary": str,  
 "intermediate\_artifacts": dict  
}

**3. Input and Output Example**

**Input:**

{  
 "analysis": {  
 "trend": "downward",  
 "drop\_detected": True,  
 "drop\_period": "2025-02",  
 "possible\_causes": ["unusual weather"],  
 "analysis\_notes": "Significant drop in February; correlates with reported snowstorm."  
 },  
 "anomalies": [  
 {"month": "2025-02", "value": 0.76, "z\_score": -2.3}  
 ],  
 "raw\_data": [  
 {"month": "2025-01", "efficiency": 0.82},  
 {"month": "2025-02", "efficiency": 0.76},  
 {"month": "2025-03", "efficiency": 0.79}  
 ],  
 "report\_params": {"output\_format": "PDF", "include\_tables": True}  
}

**Intermediate Output:**

{  
 "plots": ["efficiency\_trend.png", "anomaly\_highlight.png"],  
 "tables": [  
 {"Month": "2025-01", "Efficiency": 0.82},  
 {"Month": "2025-02", "Efficiency": 0.76},  
 {"Month": "2025-03", "Efficiency": 0.79}  
 ],  
 "narrative": "Efficiency dropped in Feb 2025, likely due to severe weather conditions."  
}

**Final Output:**

{  
 "report\_html": "<html>...narrative, plots, tables...</html>",  
 "plots": ["efficiency\_trend.png", "anomaly\_highlight.png"],  
 "tables": [  
 {"Month": "2025-01", "Efficiency": 0.82},  
 {"Month": "2025-02", "Efficiency": 0.76},  
 {"Month": "2025-03", "Efficiency": 0.79}  
 ],  
 "summary": "Efficiency dropped in Feb 2025, likely due to severe weather conditions.",  
 "intermediate\_artifacts": {  
 "plots": ["efficiency\_trend.png", "anomaly\_highlight.png"],  
 "tables": [...],  
 "narrative": "..."  
 }  
}

**4. What Is the Source of Context?**

* **ManagerAgent:** Passes all analysis/anomaly results, raw data, and report parameters.
* **BigQuery:** For any additional data needed for plots/tables.

**5. What Is the Technical Flow to Get Context?**

1. Receives all inputs from ManagerAgent.
2. Generates plots/tables (Plotly, Matplotlib).
3. Formats report (Jinja2, WeasyPrint).
4. Returns report, plots, tables, and narrative to ManagerAgent.

**6. Summary and Best Practices**

* Modularize templates and plotting code for reuse.
* Provide both final output and intermediate artifacts for debugging or further processing.
* **Integration:** ManagerAgent → ReportGenerationAgent (with all inputs), returns report to ManagerAgent.

D**etailed, commented, and documented code template** for all three modules, with hardcoded input values provided at every step for clarity and easy testing.

**Module 1: Query Understanding Agent & Feedback Agent**

# agents/module1\_query\_feedback.py  
  
class QueryUnderstandingAgent:  
 """  
 Parses user queries into structured query objects using provided context.  
 """  
 def parse(self, user\_query: str, context: dict) -> dict:  
 # Example of hardcoded input values for demonstration  
 print(f"Parsing user query: {user\_query}")  
 print(f"Using context: {context}")  
 # Hardcoded output for the example query  
 return {  
 "intent": "causal\_report",  
 "entities": {"country": "US"},  
 "metrics": ["efficiency"],  
 "dimensions": ["month"],  
 "filters": {"country": "US"},  
 "output\_type": "report\_with\_plots\_tables",  
 "context\_used": {  
 "previous\_queries": ["Show me efficiency for US in 2024", "Compare with 2023"],  
 "last\_feedback": "Wanted more details on weather impact"  
 },  
 "confidence": 0.97,  
 "clarification\_needed": False  
 }  
  
class FeedbackAgent:  
 """  
 Logs user feedback and updates context/memory.  
 """  
 def log\_feedback(self, feedback\_data: dict) -> dict:  
 print(f"Logging feedback: {feedback\_data}")  
 # Hardcoded output for demonstration  
 return {  
 "status": "success",  
 "message": "Feedback recorded and context updated.",  
 "feedback\_id": "fb\_00123",  
 "context\_updated": True  
 }

**Module 2: Semantic Layer & Data Retrieval Agent**

# agents/module2\_data\_retrieval.py  
  
class SemanticLayer:  
 """  
 Maps structured queries to SQL using business logic/config.  
 """  
 def generate\_sql(self, parsed\_query: dict) -> str:  
 print(f"Generating SQL for parsed query: {parsed\_query}")  
 # Hardcoded SQL generation for demonstration  
 country = parsed\_query["filters"]["country"]  
 sql = f"SELECT month, efficiency FROM efficiency\_table WHERE country = '{country}' ORDER BY month"  
 return sql  
  
class DataRetrievalAgent:  
 """  
 Executes SQL queries and returns tabular data.  
 """  
 def \_\_init\_\_(self):  
 self.semantic\_layer = SemanticLayer()  
  
 def fetch\_data(self, parsed\_query: dict) -> tuple:  
 sql = self.semantic\_layer.generate\_sql(parsed\_query)  
 print(f"Generated SQL: {sql}")  
 # Hardcoded data for demonstration  
 data = [  
 {"month": "2025-01", "efficiency": 0.82},  
 {"month": "2025-02", "efficiency": 0.76},  
 {"month": "2025-03", "efficiency": 0.79}  
 ]  
 return sql, data

**Module 3: Data Analysis Agent, Anomaly Detection Agent, Report Generation Agent**

# agents/module3\_analysis\_report.py  
  
class DataAnalysisAgent:  
 """  
 Performs trend and causal analysis on tabular data.  
 """  
 def analyze(self, data: list) -> dict:  
 print(f"Analyzing data: {data}")  
 # Hardcoded analysis for demonstration  
 return {  
 "trend": "downward",  
 "drop\_detected": True,  
 "drop\_period": "2025-02",  
 "possible\_causes": ["Weather anomaly"],  
 "analysis\_notes": "Significant drop in February; correlates with reported snowstorm.",  
 "intermediate\_results": {  
 "trend\_array": [-0.06, +0.03],  
 "correlations": {"weather": -0.8},  
 "mean\_efficiency": 0.79  
 }  
 }  
  
class AnomalyDetectionAgent:  
 """  
 Detects anomalies in the data using statistical or ML methods.  
 """  
 def detect(self, data: list) -> dict:  
 print(f"Detecting anomalies in data: {data}")  
 # Hardcoded anomaly detection for demonstration  
 return {  
 "anomalies": [  
 {"month": "2025-02", "value": 0.76, "z\_score": -2.3}  
 ],  
 "explanation": "Significant efficiency drop detected in Feb 2025.",  
 "intermediate\_results": {  
 "z\_scores": [0.5, -2.3, 0.8],  
 "anomaly\_flags": [False, True, False]  
 }  
 }  
  
class ReportGenerationAgent:  
 """  
 Formats analysis and anomalies into a report (HTML, plots, tables).  
 """  
 def generate(self, analysis: dict, anomalies: dict, raw\_data: list = None) -> dict:  
 print(f"Generating report with analysis: {analysis} and anomalies: {anomalies}")  
 plots = ["efficiency\_trend.png", "anomaly\_highlight.png"]  
 tables = raw\_data or []  
 narrative = analysis.get("analysis\_notes", "No notes.")  
 return {  
 "report\_html": f"<html><body><h2>Efficiency Report</h2><p>{narrative}</p></body></html>",  
 "plots": plots,  
 "tables": tables,  
 "summary": "Efficiency dropped in Feb 2025, likely due to severe weather conditions.",  
 "intermediate\_artifacts": {  
 "plots": plots,  
 "tables": tables,  
 "narrative": narrative  
 }  
 }

**Manager Agent (Integration Example)**

# agents/manager\_agent.py  
  
class ManagerAgent:  
 def \_\_init\_\_(self):  
 self.query\_agent = QueryUnderstandingAgent()  
 self.data\_agent = DataRetrievalAgent()  
 self.analysis\_agent = DataAnalysisAgent()  
 self.anomaly\_agent = AnomalyDetectionAgent()  
 self.report\_agent = ReportGenerationAgent()  
 self.feedback\_agent = FeedbackAgent()  
  
 def handle\_query(self, user\_query: str, context: dict = None) -> dict:  
 print(f"ManagerAgent received query: {user\_query}")  
 parsed\_query = self.query\_agent.parse(user\_query, context or {})  
 sql, data = self.data\_agent.fetch\_data(parsed\_query)  
 analysis = self.analysis\_agent.analyze(data)  
 anomalies = self.anomaly\_agent.detect(data)  
 report = self.report\_agent.generate(analysis, anomalies, data)  
 return {  
 "sql": sql,  
 "data": data,  
 "analysis": analysis,  
 "anomalies": anomalies,  
 "report": report  
 }  
  
 def handle\_feedback(self, feedback\_data: dict) -> dict:  
 print(f"ManagerAgent received feedback: {feedback\_data}")  
 return self.feedback\_agent.log\_feedback(feedback\_data)

**Example Usage with Hardcoded Values**

if \_\_name\_\_ == "\_\_main\_\_":  
 manager = ManagerAgent()  
 user\_query = "Can you please generate a report stating the cause of efficiency drop in the US. Please give supporting plots or tables."  
 context = {  
 "user\_id": "u123",  
 "session\_id": "sess456",  
 "previous\_queries": ["Show me efficiency for US in 2024", "Compare with 2023"],  
 "last\_feedback": "Wanted more details on weather impact",  
 "timestamp": "2025-06-26T12:41:00Z"  
 }  
  
 # Handle query  
 result = manager.handle\_query(user\_query, context)  
 print("\nFinal Report Summary:", result["report"]["summary"])  
  
 # Handle feedback  
 feedback = {  
 "query\_id": "query\_001",  
 "user\_id": "u123",  
 "session\_id": "sess456",  
 "feedback": "Good but add more weather details",  
 "timestamp": "2025-06-26T13:00:00Z"  
 }  
 feedback\_result = manager.handle\_feedback(feedback)  
 print("\nFeedback Result:", feedback\_result)