Engineering Technical Assessment Response: Email Module Improvements

# 1. System Design Overview

Architecture Diagram Description:  
- Input: Incoming Email (Plain Text / HTML / Forwarded)  
- Preprocessing Module: Cleans formatting, strips headers, identifies message body, splits attachments  
- Bid Source Handler (Abstract Factory): Determines source-specific extraction logic  
- LLM Parser: Sends cleaned content to LLM API, receives structured JSON (items, quantities, deadlines, part numbers)  
- Supplier Matcher: Matches extracted items with local supplier catalog  
- UI Module: Composes email draft with autofill suggestions/dropdowns  
  
Components:  
1. Email Ingestor (handles raw email formats)  
2. Email Cleaner/Preprocessor  
3. Bid Parser Interface (IBidParser) with source-specific implementations (e.g., SEWPBidParser, NASAParser)  
4. LLMClient (interacts with GPT model, handles prompt/response)  
5. SupplierMatcher (uses text similarity and product catalog data)  
6. UI Enhancer (injects dropdowns/autofills in email compose window)

# 2. LLM Interaction Strategy

Extraction Steps:  
- Convert email to plaintext if HTML/forwarded  
- Concatenate inline message + attachment content (text or OCR if PDF)  
- Send to LLM with structured extraction prompt:  
  
Prompt Example:  
Extract the following from this email and attachment:  
- Items Requested  
- Quantity per item  
- Part Number (if available)  
- Due Date or Submission Deadline  
Return as JSON.  
  
Expected Response:  
{  
 "items": [  
 {"name": "Cisco Switch 9300", "quantity": 2, "part\_number": "C9300-24T-E"},  
 {"name": "Fiber Optic Cable", "quantity": 10}  
 ],  
 "due\_date": "2025-07-01"  
}

# 3. Supplier Matching Logic

- Use token-based fuzzy matching (e.g., Levenshtein, cosine similarity over embeddings)  
- Rank supplier catalog entries based on match to item name and part number  
- Include threshold filter to avoid mismatches  
- Cache previous matches for repeat performance  
- Surfaced as dropdowns in email UI when composing replies

# 4. Bid Source Abstraction Design

Use Strategy + Factory Pattern to isolate logic per bid source:  
  
class IBidParser:  
 def extract\_fields(self, cleaned\_email\_text: str, attachments: List[bytes]) -> Dict:  
 pass  
  
class SEWPBidParser(IBidParser):  
 def extract\_fields(self, cleaned\_email\_text, attachments):  
 # Custom logic for SEWP structure  
 return extracted\_dict  
  
class BidParserFactory:  
 def get\_parser(source\_type: str) -> IBidParser:  
 if source\_type == "SEWP":  
 return SEWPBidParser()  
 elif source\_type == "NASA":  
 return NASAParser()  
 return DefaultBidParser()

# 5. Code Samples

Parsing and Classifying Email:  
class EmailPreprocessor:  
 def clean\_email(self, raw\_email):  
 # Strip HTML, remove headers, flatten forwarded chains  
 return clean\_text  
  
Matching Suppliers:  
class SupplierMatcher:  
 def match(self, item\_name, part\_number):  
 best\_matches = []  
 for supplier in supplier\_catalog:  
 score = fuzzy\_score(supplier.item\_name, item\_name)  
 if part\_number and part\_number in supplier.part\_numbers:  
 score += 0.3 # boost for exact part number  
 if score > 0.7:  
 best\_matches.append((supplier, score))  
 return sorted(best\_matches, key=lambda x: x[1], reverse=True)

# 6. Frontend Enhancement Mockup

UI Description:  
- In the email compose UI, when the system detects a line item, a dropdown appears inline:  
| Cisco Switch 9300 | Qty: 2 | Supplier: [▼ Select Supplier ] |  
 └─ Cisco Direct  
 └─ TechData  
 └─ CDW-G  
- Hovering shows supplier details (email, phone, past performance score)  
- Clicking auto-inserts contact into To/CC field and prefills response template