Design Choices and Architecture

Team 70 - Contributors

Name	Email Address	Contributions %
NEERAJ BHATT	2024aa05020@wilp.bits-pilani.ac.in	100%
V. S. BALAKRISHNAN	2024aa05017@wilp.bits-pilani.ac.in	100%
KURUVELLA VENKATA SAI UPENDRA	2024aa05016@wilp.bits-pilani.ac.in	100%
SAJAL CHAUDHARY	2024aa05026@wilp.bits-pilani.ac.in	100%
SACHIN KUMAR	2024aa05024@wilp.bits-pilani.ac.in	100%

1. Overall Architecture

Microservices vs Monolithic

Choice: Monolithic architecture with modular design

Rationale:

- Suitable for assignment scope and deployment constraints
- Single deployment unit simplifies testing and management

- Can be easily refactored to microservices if needed
- Reduces operational complexity

Module Structure:

- Models: Data structures and schemas
- Routes: API endpoint definitions
- Services: Business logic (QA engine, document indexing)
- Utils: Helper functions (document processing)

---

2. Backend Framework Selection

FastAPI vs Flask vs Django

Choice: FastAPI

Advantages:

- Built-in async support for better performance
- Automatic OpenAPI documentation (/docs endpoint)
- Type hints and validation with Pydantic
- Faster execution compared to Flask
- Modern and actively maintained

Comparison:

Feature	FastAPI	Flask	Django
Performance	Very Fast	Fast	Moderate
Learning Curve	Medium	Low	High
Built-in Validation	Yes	No	Yes
Auto Documentation	Yes	No	Yes
Async Support	Native	Limited	Limited

---

3. NLP Model Selection

Question Answering Model

Choice: RoBERTa-base fine-tuned on SQuAD 2.0 (deepset/roberta-base-squad2)

Model Details:

- Model Name: roberta-base
- Training Data: SQuAD 2.0 (100K+ QA pairs)
- Architecture: Transformer-based encoder
- Parameters: ~125M

Evaluation on SQuAD 2.0:

- Exact Match (EM): 89.2%
- F1 Score: 95.1%
- Handles unanswerable questions

Why RoBERTa over alternatives:

Model	Speed	Accuracy	Size	Notes
BERT	Medium	85%	110MB	Baseline transformer
RoBERTa	Medium	89%	110MB	Improved BERT training
ALBERT	Fast	87%	46MB	Smaller but lower accuracy
ELECTRA	Medium	88%	110MB	Good but less proven
FLAN-T5	Slow	92%	892MB	Larger, abstractive capable

Decision Factors:

- Good balance between accuracy and inference speed (~500ms per query)
- Proven on extractive QA tasks
- Active community support
- Easy to fine-tune if needed
- Handles out-of-context questions gracefully

---

4. Passage Retrieval Strategy

TF-IDF Similarity vs Dense Vector Search

Choice: TF-IDF with sentence windowing for MVP

Implementation Details:

- TF-IDF Vectorizer from scikit-learn
- Window size: 3 sentences
- Cosine similarity matching
- Top-K retrieval: configurable (1-5)

Advantages:

- Fast computation, no indexing overhead
- Works well with diverse document types
- Interpretable results
- Lightweight deployment

Limitations:

- Doesn't capture semantic meaning
- Performance degrades with large collections
- Requires exact term matching

Future Enhancement: Dense vector embeddings (SBERT, DPR)

Why not Elasticsearch/Solr:

- Adds operational complexity
- Overkill for MVP scope
- Additional deployment requirements

---

5. Confidence Scoring Mechanism

Combined Scoring Formula

Choice: Weighted average of similarity and QA confidence

Formula:

```
confidence_score = 0.3 × similarity_score + 0.7 × qa_score
```

Scoring Details:

TF-IDF Similarity Score:

- Range: 0 to 1

- Measures passage relevance to question
- Higher weight in formula: 0.3

QA Model Confidence:

- Range: 0 to 1
- Output from transformer model
- Higher weight in formula: 0.7

Rationale for Weights:

- QA model's confidence (0.7) is more reliable indicator
- Passage relevance (0.3) provides additional validation
- Prevents high scores for irrelevant passages with high QA confidence

Example:

```
Question: "What is AI?"
Passage A (relevant): TF-IDF=0.8, QA=0.9 → Combined=0.87
Passage B (irrelevant): TF-IDF=0.2, QA=0.95 → Combined=0.54
```

Passage A ranks higher despite lower QA score due to better relevance.

Alternative Approaches Considered:

1. Maximum: max(similarity, qa_score) - Ignores dimension
2. Multiplication: similarity × qa_score - Penalizes partial matches
3. Simple average: (similarity + qa_score) / 2 - Equal weighting

---

6. Document Storage Strategy

In-Memory vs Database

Choice: In-memory storage (v1) → Database for production

Current Implementation:

- Python dictionary for document storage
- UUID for unique document IDs
- Simple metadata tracking

Data Structure:

```
documents = {
    "doc_id": {
        "filename": "document.pdf",
        "upload_time": "ISO format",
        "text": "Full document text",
        "passages": [(text, start, end), ...],
        "text_length": 5000,
        "num_passages": 100
    }
}
```

Advantages:

- Zero latency access
- Simple implementation
- No database maintenance
- Suitable for MVP

Limitations:

- Data lost on restart
- Doesn't scale beyond 100+ documents
- No persistence
- Single-thread limitations

Migration Path to Database:

```
# SQLAlchemy models
class Document(Base):
    id: str
    filename: str
    upload_time: datetime
    text: str
    num_passages: int

# With async database access
async def add_document(session, doc):
    session.add(doc)
    await session.commit()
```

---

7. Frontend Technology Stack

Single Page App vs Traditional Multi-page

Choice: Static HTML with Vanilla JavaScript

Technology Rationale:

Aspect	Choice	Alternative	Reason
Framework	Vanilla JS	React/Vue	No build process needed
Styling	Bootstrap 5	Tailwind/Custom	Fast prototyping
HTTP	Fetch API	Axios/jQuery	Modern, built-in
Storage	SessionStorage	IndexedDB	Simple state management

No Framework Benefits:

- No npm/build pipeline complexity
- Can run directly from file system
- Minimal dependencies
- Easy to understand and modify

Frontend Architecture:

```
HTML (structure)
  ↓
CSS (styling)
  ↓
JavaScript (behavior)
  ↓
Fetch API (backend communication)
```

---

8. Error Handling Strategy

Graceful Degradation

Approach: Comprehensive error handling at each layer

Backend Layer:

```
try:
    # Process document
    text = DocumentProcessor.extract_text(file_path)
except ValueError as e:
    raise HTTPException(status_code=400, detail=str(e))
except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))
```

Frontend Layer:

```
try {
    const response = await fetch(url);
    if (!response.ok) throw new Error(`HTTP ${response.status}`);
    return await response.json();
} catch (error) {
    showError(error.message);
    logError(error);
}
```

User-Facing Errors:

- Clear, non-technical messages

- Actionable suggestions
- Auto-dismissing notifications

---

9. API Design Principles

RESTful vs GraphQL

Choice: RESTful API

Endpoints Design:

```
/api/documents
  ├── GET  /list           - List documents
  ├── POST /upload         - Upload document
  └── DELETE /{id}         - Delete document

/api/qa
  ├── POST /ask            - Ask on uploaded docs
  ├── POST /ask-direct     - Ask on direct text
  └── GET /health          - Health check
```

Why REST:

- Simpler to understand and implement
- Standard HTTP methods
- Well-known conventions
- Sufficient for current scope

API Versioning: Not included in v1 (can be added as /api/v1/)

---

10. Security Considerations

Current Implementation

CORS: Enabled for all origins (suitable for development)

File Upload:

- Extension validation (whitelist: PDF, DOCX, TXT)
- Temporary file storage in /tmp
- Files deleted after processing
- No file size limit yet (add for production)

Input Validation:

- Pydantic models for all inputs
- Question length limits (512 chars)
- Empty check for file uploads

Production Recommendations

1. Implement authentication (JWT tokens)
2. Add rate limiting (10 req/min per IP)
3. File upload size limit (50MB)
4. Input sanitization for XSS prevention
5. HTTPS/TLS for data in transit
6. Database encryption at rest

---

11. Performance Optimization

Caching Strategy

Implemented: None in v1

Potential Optimizations:

```
# Query result caching
from functools import lru_cache

@lru_cache(maxsize=100)
def get_document_passages(doc_id: str):
    return indexer.get_document_passages(doc_id)
```

Model Loading

Approach: Lazy loading on first use

- Model downloaded on first QA request
- Cached for subsequent requests
- Typical load time: 5-10 seconds

Batch Processing

Future Enhancement: Process multiple questions in parallel

```
# Async question processing
async def process_questions(questions: List[str]):
    tasks = [answer_question(q) for q in questions]
    return await asyncio.gather(*tasks)
```

12. Testing Strategy

Unit Testing

```
def test_document_processor():
    text = "Sample text. Another sentence."
    sentences = DocumentProcessor.split_into_sentences(text)
    assert len(sentences) == 2

def test_qa_engine():
    qa_engine = QAEngine()
    result = qa_engine.answer_question(
        "What is AI?",
        "AI is artificial intelligence"
    )
    assert "artificial intelligence" in result["answer"]
```

Integration Testing

- Upload → Ask → Verify answer flow
- Multiple document handling
- Error condition handling

Load Testing

- Simulate concurrent users
- Measure response times
- Monitor memory usage

13. Deployment Considerations

Development

```
python run.py  # Local testing with reload enabled
```

Production

```
gunicorn -w 4 -b 0.0.0.0:8000 app.main:app  # Multi-worker setup
```

Containerization (Optional)

```
FROM python:3.10-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0"]
```

---

14. Scalability Path

Current Limitations

- Single-threaded document processing
- In-memory storage
- No load balancing

Migration Path

1. **Database**: PostgreSQL for persistence
2. **Vector Search**: FAISS or Pinecone for semantic search
3. **Queue System**: Celery for async processing
4. **Caching**: Redis for query result caching
5. **Containerization**: Docker + Kubernetes orchestration

Projected Architecture (Production)

```
Load Balancer (nginx)
    ↓
[API 1] [API 2] [API 3]  (Horizontal scaling)
    ↓
PostgreSQL (Persistent storage)
Redis (Caching)
Celery Workers (Async tasks)
Vector DB (Semantic search)
```

---

15. Lessons Learned & Tradeoffs

Simplicity vs Completeness

Chosen: Simplicity for MVP

- Extractive QA only (not abstractive)
- Basic TF-IDF retrieval (not semantic)
- In-memory storage (not persistent)

Speed vs Accuracy

Chosen: Balanced approach

- RoBERTa offers 89% accuracy, ~500ms inference
- Could use FLAN-T5 for 92% but 2-3 second inference
- Combined scoring balances both factors

User Experience vs Implementation

Chosen: Good UX with reasonable implementation effort

- Bootstrap for quick, professional styling
- Real-time validation and feedback
- Clear error messages

Summary of Key Decisions

Aspect	Choice	Reasoning
Framework	FastAPI	Modern, fast, good docs
QA Model	RoBERTa-SQuAD2	Accuracy-speed balance
Retrieval	TF-IDF	Fast, lightweight, works
Storage	In-memory	Simple MVP, can scale
Frontend	Vanilla JS	No build pipeline
API Style	REST	Simple, proven
Scoring	Combined	Balanced relevance

These choices prioritize **simplicity and functionality** for the assignment while maintaining **paths to scale** for production.