Model Specialization Details - Requirements Extraction

Version: 1.0
Date: November 2024

Executive Summary

The models used are GENERIC models, NOT fine-tuned specifically for requirements extraction.

The specialization is achieved through **prompt engineering** - carefully crafted system prompts and user prompts that guide the generic models to perform requirements extraction tasks.

Model Type: Generic vs. Fine-Tuned

Current Implementation: Generic Models

The system uses **generic, pre-trained models** that are specialized through **prompt engineering**:

Model Type	Model Name	Specialization Method
OpenAI GPT	gpt-4o-mini, gpt-4o, gpt-3.5-turbo	Prompt Engineering
Ollama	llama3.2, mistral, etc.	Prompt Engineering

What This Means

- **Generic Models**: Pre-trained on general text data, not specifically trained on requirements extraction
- **Prompt Engineering**: Specialized behavior through carefully designed prompts
- **No Fine-Tuning**: Models are not fine-tuned on requirements extraction datasets
- **No Custom Training**: No domain-specific training data used

---

How Generic Models Are Specialized

1. System Prompt (Role Definition)

The system prompt defines the AI's role and expertise:

For OpenAI GPT Models

```
system_prompt = """You are an expert business analyst who extracts requirements
from meeting discussions. Extract functional requirements, non-functional
requirements, constraints, assumptions, and action items. Return structured JSON."""
```

Location: requirements_extractor.py, line 297

For Ollama Models

```
system_prompt = """You are an expert business analyst who extracts requirements
from meeting discussions. Extract functional requirements, non-functional
requirements, constraints, assumptions, and action items. Return structured JSON."""
```

Location: requirements_extractor.py, line 259

Key Elements:

- **Role Definition**: "expert business analyst"
- **Task Specification**: "extracts requirements from meeting discussions"
- **Output Format**: "Return structured JSON"

---

2. User Prompt (Detailed Instructions)

The user prompt provides comprehensive extraction instructions:

```
prompt = """Analyze the following meeting transcript and extract all requirements,
decisions, and action items.

Meeting Transcript:
{conversation}

Please extract and structure the following information in JSON format:
1. **Functional Requirements**: Features, functionalities, and capabilities discussed
2. **Non-Functional Requirements**: Performance, security, usability, scalability requirements
3. **Business Rules**: Rules, constraints, and business logic mentioned
4. **Assumptions**: Any assumptions made during the discussion
5. **Action Items**: Tasks assigned with owners and deadlines if mentioned
6. **Decisions**: Key decisions made during the meeting
7. **Stakeholders**: People mentioned and their roles/interests

For each requirement, include:
- ID (auto-generated)
- Description
- Priority (if mentioned: High/Medium/Low)
- Source speaker
- Related discussion context

Return the result as a JSON object with the following structure:
{
  "functional_requirements": [
    {
      "id": "FR-001",
      "description": "...",
      "priority": "High/Medium/Low",
      "speaker": "...",
      "context": "..."
    }
  ],
  "non_functional_requirements": [...],
  "business_rules": [...],
  "assumptions": [...],
```

```
 "action_items": [
   {
     "id": "AI-001",
     "task": "...",
     "owner": "...",
     "deadline": "...",
     "status": "Open"
   }
 ],
 "decisions": [
   {
     "id": "D-001",
     "decision": "...",
     "rationale": "...",
     "decision_maker": "..."
   }
 ],
 "stakeholders": [
   {
     "name": "...",
     "role": "...",
     "interests": "..."
   }
 ]
}"""
```

Location: requirements_extractor.py, function _create_extraction_prompt(), lines 331-392

Key Elements:

- **Task Definition**: Clear instruction to analyze and extract
- **Categories**: Specific list of what to extract
- **Structure**: Detailed JSON schema with examples
- **Format Requirements**: Explicit field requirements

---

3. Configuration Parameters

Temperature Setting

```
temperature=0.3
```

Purpose: Lower temperature (0.3) makes output more deterministic and consistent

Effect:

- More consistent extraction patterns
- Less creative/varied responses
- Better adherence to JSON schema
- More reliable structured output

Location:

- OpenAI: requirements_extractor.py, line 305

- Ollama: requirements_extractor.py, line 270

Response Format Enforcement (OpenAI Only)

```
response_format={"type": "json_object"}
```

Purpose: Forces OpenAI models to return valid JSON

Effect:

- Guarantees JSON output
- Reduces parsing errors
- More reliable structure

Location: requirements_extractor.py, line 304

Complete Prompt Structure

For OpenAI GPT Models

```
messages = [
  {
    "role": "system",
    "content": "You are an expert business analyst who extracts requirements
          from meeting discussions. Extract functional requirements,
          non-functional requirements, constraints, assumptions, and
          action items. Return structured JSON."
  },
  {
    "role": "user",
    "content": """
    Analyze the following meeting transcript and extract all requirements...

    [Full detailed prompt with JSON schema]
    """
  }
]

response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=messages,
  response_format={"type": "json_object"},
  temperature=0.3
)
```

For Ollama Models

```
system_prompt = "You are an expert business analyst..."
full_prompt = f"{system_prompt}\n\n{prompt}\n\nIMPORTANT: Return ONLY valid JSON, no other text."

response = requests.post(
    "http://localhost:11434/api/generate",
    json={
        "model": "llama3.2",
        "prompt": full_prompt,
        "stream": False,
        "options": {
            "temperature": 0.3
        }
    }
)
```

Why Generic Models Work

1. Strong General Capabilities

Modern LLMs (GPT-4, GPT-3.5, Llama) are trained on vast amounts of text, including:

- Business documents
- Technical specifications
- Meeting transcripts
- Requirements documents
- Project documentation

This general training gives them the ability to understand:

- Business terminology
- Technical concepts
- Structured information
- Context and relationships

2. Instruction Following

These models excel at:

- Following detailed instructions
- Understanding structured formats
- Extracting specific information
- Formatting output as requested

3. Few-Shot Learning Through Prompts

The detailed prompt acts as a "few-shot" example:

- Shows the desired output format
- Provides examples of structure
- Defines the task clearly
- Specifies all requirements

Advantages of Generic Models + Prompt Engineering

Flexibility

- **Easy Updates**: Change prompts without retraining
- **Quick Iteration**: Test different prompt variations
- **No Training Data**: Don't need labeled datasets
- **No Compute**: No fine-tuning infrastructure needed

Cost-Effective

- **No Training Costs**: Use pre-trained models as-is
- **No Data Collection**: Don't need to gather training data
- **No Annotation**: Don't need human labelers
- **Immediate Use**: Start using immediately

Maintainability

- **Transparent**: Prompts are visible and editable
- **Version Control**: Easy to track prompt changes
- **A/B Testing**: Easy to test different prompts
- **Debugging**: Can see exactly what instructions are given

Generalization

- **Works Across Domains**: Same model for different industries
- **Adaptable**: Can adjust prompts for different use cases
- **No Overfitting**: Generic models don't overfit to specific data

---

Limitations of Generic Models

⚠ Prompt Dependency

- **Sensitive to Wording**: Small prompt changes can affect output
- **Requires Tuning**: May need prompt iteration for best results
- **Context Limits**: Limited by model's context window

⚠ Consistency

- **Variability**: May produce slightly different outputs
- **Edge Cases**: May miss domain-specific nuances
- **Quality**: Depends on model's general capabilities

⚠ No Domain-Specific Training

- **Generic Knowledge**: Not trained on requirements-specific data
- **No Specialized Patterns**: Doesn't learn from requirements datasets
- **General Understanding**: Relies on general language understanding

---

Comparison: Generic vs. Fine-Tuned Models

Aspect	Generic + Prompts (Current)	Fine-Tuned Model
Training	Pre-trained only	Pre-trained + fine-tuned
Training Data	None needed	Requires labeled dataset
Training Time	None	Hours to days
Training Cost	None	Significant compute
Flexibility	High (change prompts)	Low (retrain needed)
Domain Specificity	Medium	High
Accuracy	Good	Potentially better
Maintenance	Easy (update prompts)	Hard (retrain)
Time to Deploy	Immediate	Weeks to months
Cost	API usage only	Training + API usage

Could We Use Fine-Tuned Models?

Yes, But...

Fine-tuning would require:

1. **Training Dataset**
    - Thousands of meeting transcripts
    - Labeled requirements (functional, non-functional, etc.)
    - Action items, decisions, stakeholders
    - High-quality annotations
2. **Training Infrastructure**
    - GPU compute (hours to days)
    - Fine-tuning framework (OpenAI, Hugging Face)
    - Model storage and versioning
3. **Ongoing Maintenance**
    - Retraining when requirements change
    - Dataset updates
    - Model version management

When Fine-Tuning Makes Sense

High Volume: Processing thousands of transcripts daily
Domain Specific: Very specialized requirements (e.g., medical, legal)
Consistency Critical: Need exact same output every time
Cost Optimization: Fine-tuning can reduce API costs at scale
Custom Patterns: Need to learn organization-specific patterns

Current Approach is Better When

Flexibility Needed: Requirements change frequently
Low to Medium Volume: Processing occasional transcripts
General Use: Works across different industries

Quick Deployment: Need to start immediately
Low Maintenance: Want easy updates

Prompt Engineering Techniques Used

1. Role-Based Prompting

```
"You are an expert business analyst..."
```

Technique: Define the AI's persona and expertise
Effect: Model adopts the perspective and knowledge of a business analyst

2. Structured Output Prompting

```
"Return the result as a JSON object with the following structure: {...}"
```

Technique: Provide explicit output schema
Effect: Model follows the exact structure specified

3. Example-Based Prompting

```
{
  "id": "FR-001",
  "description": "...",
  ...
}
```

Technique: Show examples of desired output
Effect: Model learns the format and style

4. Instruction Chaining

```
1. Analyze transcript
2. Extract requirements
3. Structure as JSON
```

Technique: Break down task into steps
Effect: Model follows logical sequence

5. Constraint Specification

```
"IMPORTANT: Return ONLY valid JSON, no other text."
```

Technique: Explicit constraints
Effect: Model adheres to strict requirements

Model Configuration Details

OpenAI GPT Models

Model Selection

Default: gpt-4o-mini

- **Parameters**: ~7 billion
- **Context Window**: 128,000 tokens
- **Cost**: $0.15/$0.60 per 1K tokens (input/output)
- **Speed**: Fast
- **Accuracy**: High

Alternative: gpt-4o

- **Parameters**: ~1.7 trillion (estimated)
- **Context Window**: 128,000 tokens
- **Cost**: $2.50/$10.00 per 1K tokens
- **Speed**: Medium
- **Accuracy**: Very High

Alternative: gpt-3.5-turbo

- **Parameters**: ~175 billion
- **Context Window**: 16,000 tokens
- **Cost**: $0.50/$1.50 per 1K tokens
- **Speed**: Fastest
- **Accuracy**: Good

API Configuration

```
{
  "model": "gpt-4o-mini",
  "messages": [...],
  "response_format": {"type": "json_object"},
  "temperature": 0.3,
  "max_tokens": null  # No limit (uses context window)
}
```

Location: requirements_extractor.py, lines 292-306

Ollama Models

Model Selection

Default: llama3.2

- **Parameters**: 3 billion
- **Context Window**: 128,000 tokens
- **Cost**: Free (local)
- **Speed**: 5-20 seconds/chunk (CPU)
- **Accuracy**: Good

Alternative: mistral

- **Parameters**: 7 billion
- **Context Window**: 8,000 tokens
- **Cost**: Free (local)
- **Speed**: 10-30 seconds/chunk (CPU)

- **Accuracy**: Very Good

API Configuration

```
{
  "model": "llama3.2",
  "prompt": "...",
  "stream": False,
  "options": {
    "temperature": 0.3
  }
}
```

Location: requirements_extractor.py, lines 263-273

Prompt Evolution & Optimization

Current Prompt Structure

The prompt has been designed to:

1. **Be Specific**: Clear instructions on what to extract
2. **Provide Examples**: JSON schema shows exact format
3. **Set Constraints**: Temperature and format enforcement
4. **Define Role**: System prompt sets context

Potential Improvements

1. Few-Shot Examples

Add example transcript → requirements pairs:

```
Example 1:
Transcript: "We need user authentication with 2FA"
Extracted: {
 "functional_requirements": [{
   "id": "FR-001",
   "description": "User authentication with two-factor authentication",
   ...
 }]
}

Now extract from this transcript: [actual transcript]
```

2. Domain-Specific Instructions

For specialized domains:

```
For healthcare requirements, also extract:
- HIPAA compliance requirements
- Patient privacy considerations
- Regulatory requirements
```

3. Quality Guidelines

```
Extraction Guidelines:
- Be specific and actionable
- Avoid vague requirements
- Include acceptance criteria when mentioned
- Link related requirements
```

4. Error Prevention

```
Common Mistakes to Avoid:
- Don't confuse action items with requirements
- Don't include questions as requirements
- Don't duplicate similar requirements
```

Testing & Validation

How to Verify Model Behavior

1. Test with Known Transcripts

Use transcripts with known requirements and verify extraction accuracy.

2. A/B Testing Prompts

Test different prompt variations:

- Different system prompts
- Different JSON schemas
- Different instruction styles

3. Output Quality Metrics

Measure:

- **Completeness**: All requirements extracted?
- **Accuracy**: Requirements correctly categorized?
- **Structure**: JSON format correct?
- **Consistency**: Similar inputs → similar outputs?

4. Edge Case Testing

Test with:

- Very short transcripts
- Very long transcripts

- Ambiguous requirements
- Multiple speakers
- Technical jargon

Code References

Key Files

1. **requirements_extractor.py**

   - System prompt: Line 259 (Ollama), Line 297 (OpenAI)
   - User prompt: Lines 331-392 (_create_extraction_prompt())
   - Model configuration: Lines 263-273 (Ollama), Lines 292-306 (OpenAI)

2. **app.py**

   - Model selection: Lines 1058-1063
   - Extraction call: Lines 1178-1186

Key Functions

- RequirementsExtractor.__init__(): Model initialization
- RequirementsExtractor.extract_requirements(): Main extraction logic
- RequirementsExtractor._create_extraction_prompt(): Prompt generation
- RequirementsExtractor._format_conversation(): Transcript formatting

---

Summary

Key Points

1. **Generic Models**: Uses standard pre-trained models (GPT, Llama)
2. **Prompt Engineering**: Specialization through carefully crafted prompts
3. **No Fine-Tuning**: Models are not fine-tuned on requirements data
4. **Flexible Approach**: Easy to update and modify
5. **Cost-Effective**: No training costs or infrastructure needed

Why This Works

- Modern LLMs have strong general capabilities
- Prompt engineering provides sufficient guidance
- Structured output enforcement ensures consistency
- Temperature control reduces variability

When to Consider Fine-Tuning

- Processing very high volumes
- Need domain-specific patterns
- Require maximum accuracy
- Have labeled training data
- Cost optimization at scale

Appendix

A. Full Prompt Example

See requirements_extractor.py, function _create_extraction_prompt() for the complete prompt.

B. Model API Documentation

- **OpenAI**: https://platform.openai.com/docs/api-reference
- **Ollama**: https://github.com/ollama/ollama/blob/main/docs/api.md

C. Prompt Engineering Resources

- OpenAI Prompt Engineering Guide
- LangChain Prompt Templates
- Anthropic Prompt Engineering

End of Model Specialization Details