Structured Output - Intermediate Handout

Machine Learning for Smarter Innovation

1 Structured Output - Intermediate Handout

Target Audience: Practitioners with Python knowledge **Duration**: 60 minutes reading + coding **Level**: Intermediate (implementation focused)

1.1 Setup

```python
import os
import json
import time
import logging
import hashlib
from datetime import datetime
from typing import List, Dict, Optional, Any
from functools import lru_cache
from dataclasses import dataclass

# Pydantic for validation
from pydantic import BaseModel, Field, validator, ValidationError

# API clients (choose one)
from anthropic import Anthropic
# from openai import OpenAI

import warnings
warnings.filterwarnings('ignore')

# Logging setup
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment
# client = Anthropic(api_key=os.environ.get("ANTHROPIC_API_KEY"))
```

This handout covers practical implementation of structured output systems that extract reliable, validated data from unstructured text using large language models. Structured output bridges the gap between natural language understanding and programmatic data processing. The techniques apply across document extraction, form processing, content analysis, and API response parsing where consistent data formats are required.

1.2 1. Function Calling for Structured Extraction

1.2.1 Concept Overview

Function calling provides a standardized interface for requesting structured outputs from language models. Instead of prompting the model to return JSON and hoping it complies, function calling defines an explicit schema that the model must follow. The model decides whether to invoke the function and generates arguments matching the defined parameter types. This approach achieves significantly higher reliability than free-form JSON generation.

1.2.2 Implementation: Basic Function Calling

```
def create_extraction_function(name: str, description: str,
                               properties: Dict[str, Dict],
                               required: List[str] = None) -> Dict:
    """Create an OpenAI-style function definition."""
    return {
        "name": name,
        "description": description,
        "parameters": {
            "type": "object",
            "properties": properties,
            "required": required or list(properties.keys())
        }
    }


# Example: Restaurant review extraction
review_function = create_extraction_function(
    name="extract_review_data",
    description="Extract structured data from a restaurant review",
    properties={
        "rating": {
            "type": "integer",
            "description": "Overall rating from 1-5",
            "minimum": 1,
            "maximum": 5
        },
        "food_quality": {
            "type": "integer",
            "description": "Food quality rating from 1-5",
            "minimum": 1,
            "maximum": 5
        },
        "service_quality": {
            "type": "integer",
            "description": "Service quality rating from 1-5",
            "minimum": 1,
            "maximum": 5
        },
        "price_level": {
            "type": "string",
            "enum": ["cheap", "moderate", "expensive"],
            "description": "Price category"
        },
        "avg_price_per_person": {
            "type": "number",
            "description": "Average price per person in dollars"
        },
        "themes": {
            "type": "array",
```

```
            "items": {"type": "string"},
            "description": "Main themes or categories",
            "maxItems": 5
        }
    },
    required=["rating", "food_quality", "service_quality", "price_level"]
)

print(json.dumps(review_function, indent=2))


class FunctionCallingClient:
    """Client wrapper for function calling APIs."""

    def __init__(self, provider: str = "anthropic"):
        self.provider = provider

    def call_with_function(self, prompt: str, function_def: Dict,
                           system_prompt: str = None) -> Dict:
        """Call API with function definition and return parsed result."""

        # Simulated API call - replace with actual implementation
        # For OpenAI:
        # response = openai.ChatCompletion.create(
        #     model="gpt-4",
        #     messages=[
        #         {"role": "system", "content": system_prompt or "Extract data
    accurately."},
        #         {"role": "user", "content": prompt}
        #     ],
        #     functions=[function_def],
        #     function_call={"name": function_def["name"]}
        # )
        # return json.loads(response.choices[0].message.function_call.
    arguments)

        # Placeholder response
        return {
            "rating": 5,
            "food_quality": 5,
            "service_quality": 4,
            "price_level": "expensive",
            "avg_price_per_person": 45.0,
            "themes": ["italian", "romantic", "authentic"]
        }


# Usage
client = FunctionCallingClient()
review_text = """
The food was absolutely amazing! Best Italian I've had in years.
Service was friendly and attentive. A bit pricey at $45 per person
but definitely worth it for a special occasion.
"""

result = client.call_with_function(review_text, review_function)
print(json.dumps(result, indent=2))
```

1.3 2. Pydantic Schema Validation

1.3.1 Concept Overview

Pydantic provides runtime type validation for Python, ensuring extracted data conforms to expected types, ranges, and formats. By defining Pydantic models that mirror your function schemas, you add a validation layer that catches malformed outputs before they enter your application. Pydantic also generates JSON schemas automatically, enabling seamless integration with function calling APIs.

1.3.2 Implementation: Type-Safe Data Models

```
class ReviewData(BaseModel):
    """Validated model for restaurant review extraction."""

    rating: int = Field(..., ge=1, le=5, description="Overall rating")
    food_quality: int = Field(..., ge=1, le=5, description="Food quality
    rating")
    service_quality: int = Field(..., ge=1, le=5, description="Service quality
     rating")
    price_level: str = Field(..., description="Price category")
    avg_price_per_person: Optional[float] = Field(None, gt=0, description="
    Average price per person")
    themes: List[str] = Field(default_factory=list, max_length=5)
    recommended_for: List[str] = Field(default_factory=list)
    confidence: Optional[float] = Field(None, ge=0, le=1)

    @validator('price_level')
    def validate_price_level(cls, v):
        """Ensure price level is one of allowed values."""
        allowed = ['cheap', 'moderate', 'expensive']
        if v.lower() not in allowed:
            raise ValueError(f"price_level must be one of {allowed}")
        return v.lower()

    @validator('recommended_for', each_item=True)
    def validate_recommendations(cls, v):
        """Validate recommendation categories."""
        allowed = ['date', 'family', 'business', 'friends', 'solo']
        if v.lower() not in allowed:
            raise ValueError(f"recommendation must be one of {allowed}")
        return v.lower()

    class Config:
        # Generate schema for function calling
        schema_extra = {
            "example": {
                "rating": 4,
                "food_quality": 5,
                "service_quality": 4,
                "price_level": "moderate",
                "avg_price_per_person": 35.0,
                "themes": ["italian", "cozy"]
            }
        }


class ProductReview(BaseModel):
    """Model for e-commerce product review extraction."""

    product_rating: int = Field(..., ge=1, le=5)
    quality_score: int = Field(..., ge=1, le=5)
```

```
    value_score: int = Field(..., ge=1, le=5)
    would_recommend: bool
    pros: List[str] = Field(default_factory=list, max_length=5)
    cons: List[str] = Field(default_factory=list, max_length=5)
    sentiment: str = Field(..., description="positive, negative, or neutral")

    @validator('sentiment')
    def validate_sentiment(cls, v):
        allowed = ['positive', 'negative', 'neutral']
        if v.lower() not in allowed:
            raise ValueError(f"sentiment must be one of {allowed}")
        return v.lower()


def validate_extraction(data: Dict, model_class: type) -> Optional[BaseModel]:
    """Validate extracted data against Pydantic model."""
    try:
        validated = model_class(**data)
        return validated
    except ValidationError as e:
        logger.error(f"Validation failed: {e}")
        return None


# Test validation
test_data = {
    "rating": 5,
    "food_quality": 5,
    "service_quality": 4,
    "price_level": "expensive",
    "themes": ["italian", "romantic"]
}

validated = validate_extraction(test_data, ReviewData)
if validated:
    print("Validation passed!")
    print(validated.dict())
    print(validated.json(indent=2))
```

1.4 3. Production Extraction Pipeline

1.4.1 Concept Overview

Production systems require robust error handling, retry logic, and fallback strategies. Network failures, rate limits, and malformed responses are common. A well-designed pipeline implements exponential backoff for retries, validates outputs at multiple stages, and gracefully degrades to simpler extraction methods when AI fails. Logging and metrics enable monitoring and debugging.

1.4.2 Implementation: Robust Extraction System

```
@dataclass
class ExtractionResult:
    """Container for extraction results with metadata."""
    success: bool
    data: Optional[Dict] = None
    validated: Optional[BaseModel] = None
    method: str = "ai"
```

```
    attempts: int = 1
    duration: float = 0.0
    error: Optional[str] = None


class StructuredExtractor:
    """Production-ready structured extraction system."""

    def __init__(self, model_class: type, function_def: Dict,
                 max_retries: int = 3, cache_enabled: bool = True):
        self.model_class = model_class
        self.function_def = function_def
        self.max_retries = max_retries
        self.cache_enabled = cache_enabled
        self.cache: Dict[str, ExtractionResult] = {}
        self.metrics = ExtractionMetrics()

    def _get_cache_key(self, text: str) -> str:
        """Generate cache key from input text."""
        return hashlib.sha256(text.encode()).hexdigest()

    def extract(self, text: str, use_fallback: bool = True) ->
    ExtractionResult:
        """Extract structured data with full error handling."""
        start_time = time.time()

        # Check cache
        if self.cache_enabled:
            cache_key = self._get_cache_key(text)
            if cache_key in self.cache:
                logger.info("Cache hit")
                return self.cache[cache_key]

        # Try AI extraction with retries
        result = self._extract_with_retry(text)

        # Fall back to rules if AI fails
        if not result.success and use_fallback:
            logger.warning("AI extraction failed, using fallback")
            result = self._rule_based_extraction(text)
            result.method = "fallback"

        # Record metrics
        result.duration = time.time() - start_time
        self.metrics.record(result)

        # Cache successful results
        if self.cache_enabled and result.success:
            self.cache[cache_key] = result

        return result

    def _extract_with_retry(self, text: str) -> ExtractionResult:
        """Attempt extraction with exponential backoff."""
        last_error = None

        for attempt in range(self.max_retries):
            try:
                # Call API (simulated)
                raw_data = self._call_api(text)

                # Validate
                validated = self.model_class(**raw_data)
```

```
                return ExtractionResult(
                    success=True,
                    data=raw_data,
                    validated=validated,
                    method="ai",
                    attempts=attempt + 1
                )

            except ValidationError as e:
                last_error = f"Validation error: {e}"
                logger.warning(f"Attempt {attempt + 1} validation failed: {e}"
)

            except Exception as e:
                last_error = f"API error: {e}"
                logger.warning(f"Attempt {attempt + 1} failed: {e}")

            # Exponential backoff
            if attempt < self.max_retries - 1:
                sleep_time = 2 ** attempt
                logger.info(f"Retrying in {sleep_time} seconds...")
                time.sleep(sleep_time)

        return ExtractionResult(
            success=False,
            method="ai",
            attempts=self.max_retries,
            error=last_error
        )

    def _call_api(self, text: str) -> Dict:
        """Make API call for extraction."""
        # Simulated - replace with actual API call
        import re

        # Simple extraction simulation
        rating_match = re.search(r'(\d)\s*stars?', text.lower())
        price_match = re.search(r'\$(\d+)', text)

        return {
            "rating": int(rating_match.group(1)) if rating_match else 4,
            "food_quality": 4,
            "service_quality": 4,
            "price_level": "expensive" if price_match and int(price_match.
group(1)) > 30 else "moderate",
            "avg_price_per_person": float(price_match.group(1)) if price_match
 else None,
            "themes": []
        }

    def _rule_based_extraction(self, text: str) -> ExtractionResult:
        """Fallback rule-based extraction."""
        import re

        try:
            # Extract rating
            rating_patterns = [
                (r'(\d)\s*(?:star|out of 5)', lambda m: int(m.group(1))),
                (r'(excellent|amazing|great)', lambda m: 5),
                (r'(good|nice|decent)', lambda m: 4),
                (r'(okay|average)', lambda m: 3),
                (r'(poor|bad)', lambda m: 2),
```

```
                (r'(terrible|awful)', lambda m: 1)
            ]

            rating = 3  # Default
            for pattern, extractor in rating_patterns:
                match = re.search(pattern, text.lower())
                if match:
                    rating = extractor(match)
                    break

            # Extract price
            price_match = re.search(r'\$(\d+)', text)
            if price_match:
                price = int(price_match.group(1))
                price_level = "cheap" if price < 15 else "expensive" if price
    > 40 else "moderate"
            else:
                price_level = "moderate"
                price = None

            data = {
                "rating": rating,
                "food_quality": rating,
                "service_quality": rating,
                "price_level": price_level,
                "avg_price_per_person": float(price) if price else None,
                "themes": [],
                "confidence": 0.5
            }

            validated = self.model_class(**data)

            return ExtractionResult(
                success=True,
                data=data,
                validated=validated,
                method="fallback",
                attempts=1
            )

        except Exception as e:
            return ExtractionResult(
                success=False,
                method="fallback",
                error=str(e)
            )


class ExtractionMetrics:
    """Track extraction performance metrics."""

    def __init__(self):
        self.total = 0
        self.successes = 0
        self.failures = 0
        self.ai_successes = 0
        self.fallback_successes = 0
        self.total_duration = 0.0
        self.total_attempts = 0

    def record(self, result: ExtractionResult):
        """Record extraction result."""
        self.total += 1
```

```
        self.total_duration += result.duration
        self.total_attempts += result.attempts

        if result.success:
            self.successes += 1
            if result.method == "ai":
                self.ai_successes += 1
            else:
                self.fallback_successes += 1
        else:
            self.failures += 1

    def summary(self) -> Dict:
        """Get metrics summary."""
        return {
            "total_extractions": self.total,
            "success_rate": self.successes / self.total if self.total > 0 else
     0,
            "ai_success_rate": self.ai_successes / self.total if self.total >
    0 else 0,
            "fallback_rate": self.fallback_successes / self.total if self.
    total > 0 else 0,
            "failure_rate": self.failures / self.total if self.total > 0 else
    0,
            "avg_duration": self.total_duration / self.total if self.total > 0
     else 0,
            "avg_attempts": self.total_attempts / self.total if self.total > 0
     else 0
        }


# Usage
extractor = StructuredExtractor(
    model_class=ReviewData,
    function_def=review_function,
    max_retries=3
)

test_reviews = [
    "Amazing food! 5 stars, worth every penny at $50 per person.",
    "Decent place. Food was okay. About $20 for lunch.",
    "Terrible experience. Would not recommend."
]

for review in test_reviews:
    result = extractor.extract(review)
    print(f"Success: {result.success}, Method: {result.method}")
    if result.validated:
        print(f"  Rating: {result.validated.rating}, Price: {result.validated.
    price_level}")

print("\nMetrics:", extractor.metrics.summary())
```

1.5 4. Multi-Stage Validation

1.5.1 Concept Overview

Beyond schema validation, production systems need business logic validation to catch semantically invalid but syntactically correct outputs. A multi-stage approach validates data at the schema level, then applies

domain rules, then optionally checks confidence scores or consistency. This layered approach catches errors that simple type checking misses.

1.5.2 Implementation: Validation Pipeline

```
class ValidationPipeline:
    """Multi-stage validation for extracted data."""

    def __init__(self, model_class: type):
        self.model_class = model_class
        self.rules: List[callable] = []

    def add_rule(self, rule: callable, description: str = ""):
        """Add a business validation rule."""
        self.rules.append((rule, description))
        return self

    def validate(self, data: Dict) -> tuple[bool, List[str]]:
        """Run all validation stages."""
        errors = []

        # Stage 1: Schema validation
        try:
            validated = self.model_class(**data)
        except ValidationError as e:
            return False, [f"Schema validation failed: {e}"]

        # Stage 2: Business rules
        for rule, description in self.rules:
            try:
                if not rule(validated):
                    errors.append(f"Business rule failed: {description}")
            except Exception as e:
                errors.append(f"Rule error ({description}): {e}")

        return len(errors) == 0, errors


# Define business rules
def rating_consistency_rule(review: ReviewData) -> bool:
    """Overall rating should align with component ratings."""
    component_avg = (review.food_quality + review.service_quality) / 2
    return abs(review.rating - component_avg) <= 1.5


def price_consistency_rule(review: ReviewData) -> bool:
    """Price level should match actual price if provided."""
    if review.avg_price_per_person is None:
        return True

    price = review.avg_price_per_person
    level = review.price_level

    if level == "cheap" and price > 20:
        return False
    if level == "expensive" and price < 30:
        return False

    return True


def suspicious_pattern_rule(review: ReviewData) -> bool:
```

```
    """Detect suspiciously perfect or inconsistent reviews."""
    # All 5s is suspicious
    if review.rating == 5 and review.food_quality == 5 and review.
    service_quality == 5:
        if review.confidence and review.confidence < 0.8:
            return False

    # 5-star overall with poor component is suspicious
    if review.rating == 5 and min(review.food_quality, review.service_quality)
     < 3:
        return False

    return True


# Build validation pipeline
pipeline = ValidationPipeline(ReviewData)
pipeline.add_rule(rating_consistency_rule, "Rating consistency")
pipeline.add_rule(price_consistency_rule, "Price consistency")
pipeline.add_rule(suspicious_pattern_rule, "Suspicious patterns")

# Test validation
test_cases = [
    {"rating": 5, "food_quality": 5, "service_quality": 5, "price_level": "
    expensive"},
    {"rating": 5, "food_quality": 2, "service_quality": 4, "price_level": "
    moderate"},
    {"rating": 3, "food_quality": 3, "service_quality": 3, "price_level": "
    cheap", "avg_price_per_person": 50}
]

for case in test_cases:
    valid, errors = pipeline.validate(case)
    print(f"Valid: {valid}")
    if errors:
        print(f"  Errors: {errors}")
```

1.6 5. Testing and Quality Assurance

1.6.1 Concept Overview

Structured output systems require comprehensive testing across schema validation, API integration, and edge cases. Unit tests verify Pydantic models catch invalid data. Integration tests confirm the full pipeline handles real-world inputs. Edge case tests ensure graceful handling of unusual inputs, missing fields, and malformed responses.

1.6.2 Implementation: Test Suite

```
import pytest

class TestSchemaValidation:
    """Test Pydantic model validation."""

    def test_valid_data_accepted(self):
        """Valid data should pass validation."""
        data = {
            "rating": 4,
```

```
            "food_quality": 5,
            "service_quality": 4,
            "price_level": "moderate"
        }
        review = ReviewData(**data)
        assert review.rating == 4
        assert review.food_quality == 5

    def test_invalid_rating_rejected(self):
        """Rating outside 1-5 should be rejected."""
        with pytest.raises(ValidationError):
            ReviewData(
                rating=6,
                food_quality=4,
                service_quality=4,
                price_level="moderate"
            )

    def test_invalid_price_level_rejected(self):
        """Invalid price level should be rejected."""
        with pytest.raises(ValidationError):
            ReviewData(
                rating=4,
                food_quality=4,
                service_quality=4,
                price_level="super_expensive"
            )

    def test_optional_fields_default_correctly(self):
        """Optional fields should use defaults."""
        data = {
            "rating": 4,
            "food_quality": 4,
            "service_quality": 4,
            "price_level": "moderate"
        }
        review = ReviewData(**data)
        assert review.avg_price_per_person is None
        assert review.themes == []


class TestExtractionPipeline:
    """Test full extraction pipeline."""

    @pytest.fixture
    def extractor(self):
        return StructuredExtractor(
            model_class=ReviewData,
            function_def=review_function,
            max_retries=2
        )

    def test_successful_extraction(self, extractor):
        """Pipeline should extract valid data."""
        result = extractor.extract("Great food! 5 stars. Price was $30.")
        assert result.success
        assert result.validated is not None
        assert result.validated.rating >= 1

    def test_fallback_on_failure(self, extractor):
        """Pipeline should fall back on AI failure."""
        # Force fallback by testing with minimal input
        result = extractor.extract("", use_fallback=True)
```

```
        # Even with empty input, fallback should work
        assert result.success or result.method == "fallback"

    def test_caching_works(self, extractor):
        """Identical inputs should return cached results."""
        text = "Amazing restaurant! 5 stars."
        result1 = extractor.extract(text)
        result2 = extractor.extract(text)

        # Second call should be much faster (cached)
        assert result2.duration < result1.duration


class TestEdgeCases:
    """Test edge case handling."""

    def test_empty_input(self):
        """Empty input should be handled gracefully."""
        extractor = StructuredExtractor(ReviewData, review_function)
        result = extractor.extract("")
        # Should not crash, may succeed with defaults or fail gracefully
        assert isinstance(result, ExtractionResult)

    def test_very_long_input(self):
        """Very long input should be handled."""
        extractor = StructuredExtractor(ReviewData, review_function)
        long_text = "Great food! " * 1000
        result = extractor.extract(long_text)
        assert isinstance(result, ExtractionResult)

    def test_special_characters(self):
        """Special characters should not break extraction."""
        extractor = StructuredExtractor(ReviewData, review_function)
        text = "Food was great! $$$$ but worth it. Rating: 5/5"
        result = extractor.extract(text)
        assert isinstance(result, ExtractionResult)


# Run tests
if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

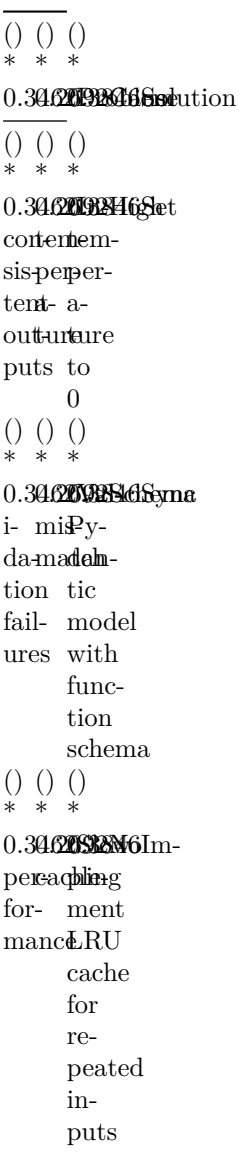
1.7 Common Parameters and Settings

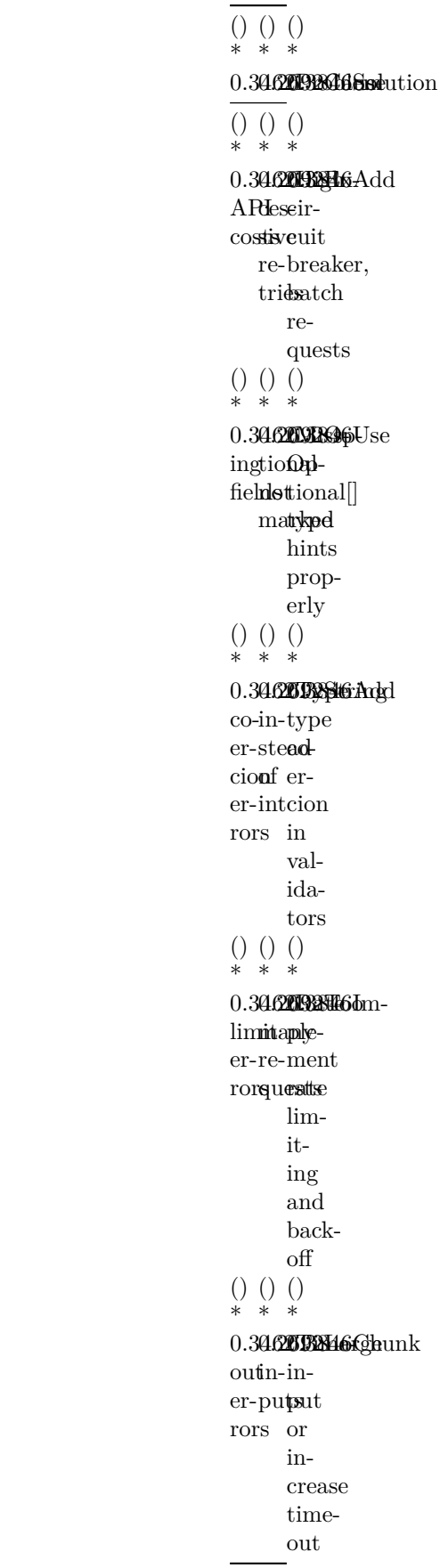
Parameter	Recommended Value	Notes
temperature	0.0 - 0.1	Lower = more consistent outputs
max_tokens	500 - 1000	Match expected output size
max_retries	3	Balance reliability vs latency
backoff_base	2	Exponential backoff multiplier
cache_ttl	3600	Cache duration in seconds
timeout	30	API call timeout in seconds
confidence_threshold	0.7	Minimum for auto-acceptance

1.8 Practice Projects

1. **Invoice Data Extractor**: Build a system that extracts vendor name, invoice number, date, line items, and total from invoice images or PDFs. Include validation for date formats and numeric consistency.
2. **Job Posting Parser**: Create an extractor for job postings that captures title, company, location, salary range, required skills, and experience level. Handle variations in how different companies format postings.
3. **Email Intent Classifier**: Develop a system that classifies emails by intent (support request, sales inquiry, complaint) and extracts key entities (product mentioned, urgency level, sentiment).
4. **Contract Clause Extractor**: Build a pipeline that identifies and extracts specific clauses from legal contracts including termination terms, liability limits, and payment schedules.

1.9 Troubleshooting





1.10 Next Steps

- Read the advanced handout for multi-model pipelines and fine-tuning
- Implement confidence scoring for extraction quality
- Build monitoring dashboards for production systems
- Explore batch processing for high-volume extraction
- Add human-in-the-loop review for low-confidence extractions

---

Structured output transforms unreliable AI responses into dependable data pipelines. The combination of explicit schemas, rigorous validation, and graceful error handling achieves reliability levels suitable for production systems. Invest in testing and monitoring to maintain quality as inputs evolve.