Handout 3: Production-Grade Structured AI Systems

Machine Learning for Smarter Innovation

1 Handout 3: Production-Grade Structured AI Systems

1.1 Advanced Techniques for 99%+ Reliability

1.1.1 Prerequisites

- Completed Handouts 1 & 2
- Production Python experience
- Understanding of system architecture
- Deployed at least one ML system

1.1.2 Part 1: Advanced Prompt Engineering

Multi-Step Reasoning

```
EXTRACTION_PROMPT = """
You are an expert data extraction system. Follow these steps:

Step 1: Read the entire review carefully
Step 2: Identify explicit mentions of quality (food, service, ambiance)
Step 3: Infer implicit ratings from sentiment
Step 4: Extract price information (exact amounts or indicators)
Step 5: Categorize the review (occasion, audience)
Step 6: Assign confidence scores for each field

Think through each step, then provide structured output.

Review: {review_text}

Reasoning (explain your thought process):
Output (JSON format):
"""
```

Few-Shot with Diverse Examples

```
FEW_SHOT_EXAMPLES = [
    {
        "review": "Meh, food was okay. Service slow. $15 lunch.",
        "output": {"rating": 3, "food_quality": 3, "service_quality": 2, "
    price_level": "cheap"}
    },
    {
        "review": "AMAZING! Best meal ever! Worth every penny of the $80.",
        "output": {"rating": 5, "food_quality": 5, "service_quality": 5, "
    price_level": "expensive"}
    },
    {
```

```
        "review": "Good Italian spot. Nothing special but solid. $25-30 range.
    ",
        "output": {"rating": 4, "food_quality": 4, "service_quality": 4, "
    price_level": "moderate"}
    }
]

def build_few_shot_prompt(review: str) -> str:
    examples_text = "\n\n".join([
        f"Example {i+1}:\nReview: {ex['review']}\nOutput: {json.dumps(ex['
    output'])}"
        for i, ex in enumerate(FEW_SHOT_EXAMPLES)
    ])

    return f"{examples_text}\n\nNow extract from:\nReview: {review}\nOutput:"
```

1.1.3 Part 2: Advanced Validation Pipeline

Custom Validation Chain

```
from typing import Protocol, List
from dataclasses import dataclass

@dataclass
class ValidationResult:
    valid: bool
    confidence: float
    errors: List[str]
    warnings: List[str]

class Validator(Protocol):
    def validate(self, data: dict) -> ValidationResult:
        ...

class SchemaValidator:
    def validate(self, data: dict) -> ValidationResult:
        errors = []
        try:
            ReviewData(**data)
        except ValidationError as e:
            errors = [str(err) for err in e.errors()]
            return ValidationResult(valid=False, confidence=0.0, errors=errors
    , warnings=[])

        return ValidationResult(valid=True, confidence=1.0, errors=[],
    warnings=[])

class BusinessRuleValidator:
    def validate(self, data: dict) -> ValidationResult:
        warnings = []
        errors = []

        # Rule 1: Rating consistency
        if data.get('rating', 0) == 5:
            if data.get('food_quality', 0) < 4 or data.get('service_quality',
    0) < 4:
                warnings.append("5-star rating but sub-par components")

        # Rule 2: Price consistency
        if data.get('price_level') == 'cheap':
            if data.get('avg_price_per_person', 0) > 20:
                errors.append("Labeled cheap but price > $20")
```

```
        # Rule 3: Confidence threshold
        if data.get('confidence', 1.0) < 0.7:
            warnings.append("Low confidence extraction")

        valid = len(errors) == 0
        confidence = 0.8 if not warnings else 0.9

        return ValidationResult(valid=valid, confidence=confidence, errors=
    errors, warnings=warnings)

class StatisticalValidator:
    """Check if values are within expected statistical distributions"""

    def __init__(self):
        # Load historical statistics
        self.rating_distribution = [0.05, 0.10, 0.20, 0.35, 0.30]  # 1-5 stars
        self.avg_price_by_level = {
            "cheap": (5, 20),
            "moderate": (20, 50),
            "expensive": (50, 200)
        }

    def validate(self, data: dict) -> ValidationResult:
        warnings = []

        # Check price alignment
        price = data.get('avg_price_per_person')
        level = data.get('price_level')

        if price and level:
            min_price, max_price = self.avg_price_by_level[level]
            if not (min_price <= price <= max_price):
                warnings.append(f"Price ${price} unusual for {level} category"
    )

        return ValidationResult(
            valid=True,
            confidence=0.9 if warnings else 1.0,
            errors=[],
            warnings=warnings
        )

class ValidationPipeline:
    """Chain multiple validators"""

    def __init__(self, validators: List[Validator]):
        self.validators = validators

    def validate(self, data: dict) -> ValidationResult:
        all_errors = []
        all_warnings = []
        min_confidence = 1.0

        for validator in self.validators:
            result = validator.validate(data)

            if not result.valid:
                return result  # Fail fast on hard errors

            all_warnings.extend(result.warnings)
            min_confidence = min(min_confidence, result.confidence)

        return ValidationResult(
```

```
            valid=True,
            confidence=min_confidence,
            errors=[],
            warnings=all_warnings
        )

# Use it
pipeline = ValidationPipeline([
    SchemaValidator(),
    BusinessRuleValidator(),
    StatisticalValidator()
])

result = pipeline.validate(extracted_data)
if result.valid and result.confidence > 0.8:
    save_to_database(extracted_data)
elif result.confidence > 0.6:
    flag_for_human_review(extracted_data, result.warnings)
else:
    reject_and_retry(extracted_data)
```

1.1.4 Part 3: Multi-Model Fallback Strategy

```
from enum import Enum

class ModelTier(Enum):
    PRIMARY = "gpt-4"
    SECONDARY = "gpt-3.5-turbo"
    FALLBACK = "claude-3-sonnet"
    RULE_BASED = "rules"

class MultiModelExtractor:
    def __init__(self):
        self.models = [
            (ModelTier.PRIMARY, self._extract_gpt4),
            (ModelTier.SECONDARY, self._extract_gpt35),
            (ModelTier.FALLBACK, self._extract_claude),
            (ModelTier.RULE_BASED, self._extract_rules)
        ]

    def extract(self, text: str) -> tuple[dict, ModelTier]:
        """Try each model until success"""

        for tier, extractor_func in self.models:
            try:
                result = extractor_func(text)

                if self._validate(result):
                    logger.info(f"Success with {tier.value}")
                    return result, tier

            except Exception as e:
                logger.warning(f"{tier.value} failed: {e}")
                continue

        raise Exception("All extraction methods failed")

    def _extract_gpt4(self, text: str) -> dict:
        return openai_extract(text, model="gpt-4", temperature=0.1)

    def _extract_gpt35(self, text: str) -> dict:
```

```
        return openai_extract(text, model="gpt-3.5-turbo", temperature=0.0)

    def _extract_claude(self, text: str) -> dict:
        return anthropic_extract(text, model="claude-3-sonnet")

    def _extract_rules(self, text: str) -> dict:
        return rule_based_extraction(text)

    def _validate(self, data: dict) -> bool:
        pipeline = ValidationPipeline([...])
        result = pipeline.validate(data)
        return result.valid and result.confidence > 0.7
```

1.1.5 Part 4: Performance Optimization

Aggressive Caching

```
import redis
import hashlib
import pickle

class CachedExtractor:
    def __init__(self, redis_url: str):
        self.redis = redis.from_url(redis_url)
        self.ttl = 3600  # 1 hour

    def _cache_key(self, text: str) -> str:
        """Generate cache key from text"""
        return f"extract:{hashlib.md5(text.encode()).hexdigest()}"

    def extract(self, text: str) -> dict:
        """Extract with caching"""

        key = self._cache_key(text)

        # Check cache
        cached = self.redis.get(key)
        if cached:
            logger.info("Cache hit")
            return pickle.loads(cached)

        # Extract
        logger.info("Cache miss, extracting")
        result = expensive_extraction(text)

        # Cache result
        self.redis.setex(key, self.ttl, pickle.dumps(result))

        return result
```

Batch Processing

```
import asyncio
from concurrent.futures import ThreadPoolExecutor

async def extract_batch(reviews: List[str], batch_size: int = 10) -> List[dict
    ]:
    """Process reviews in parallel batches"""

    results = []

    # Split into batches
```

```
    for i in range(0, len(reviews), batch_size):
        batch = reviews[i:i+batch_size]

        # Process batch in parallel
        with ThreadPoolExecutor(max_workers=batch_size) as executor:
            batch_results = list(executor.map(extract_review_data, batch))

        results.extend(batch_results)

        # Rate limiting
        await asyncio.sleep(1)

    return results

# Use it
reviews = [...]  # 1000 reviews
results = asyncio.run(extract_batch(reviews, batch_size=20))
```

Token Optimization

```
def optimize_prompt(text: str, max_tokens: int = 2000) -> str:
    """Reduce prompt size while maintaining quality"""

    # Remove unnecessary whitespace
    text = " ".join(text.split())

    # Truncate if too long
    if len(text) > max_tokens:
        text = text[:max_tokens] + "... [truncated]"

    # Remove common filler words
    fillers = ["basically", "literally", "actually", "like"]
    for filler in fillers:
        text = text.replace(f" {filler} ", " ")

    return text
```

1.1.6 Part 5: Production Monitoring

Comprehensive Metrics

```
from prometheus_client import Counter, Histogram, Gauge
import statsd

# Define metrics
extraction_requests = Counter('extraction_requests_total', 'Total extraction
    requests')
extraction_success = Counter('extraction_success_total', 'Successful
    extractions')
extraction_failures = Counter('extraction_failures_total', 'Failed extractions
    ')
extraction_duration = Histogram('extraction_duration_seconds', 'Extraction
    duration')
extraction_confidence = Histogram('extraction_confidence', 'Confidence scores'
    )
active_extractions = Gauge('active_extractions', 'Currently processing')

class MonitoredExtractor:
    def __init__(self, extractor):
        self.extractor = extractor

    def extract(self, text: str) -> dict:
```

```
        extraction_requests.inc()

        with active_extractions.track_inprogress():
            with extraction_duration.time():
                try:
                    result = self.extractor.extract(text)
                    extraction_success.inc()

                    # Track confidence
                    if 'confidence' in result:
                        extraction_confidence.observe(result['confidence'])

                    return result

                except Exception as e:
                    extraction_failures.inc()
                    raise
```

Alerting

```
from dataclasses import dataclass
from typing import Callable

@dataclass
class Alert:
    name: str
    condition: Callable
    action: Callable
    cooldown: int = 300  # 5 minutes

class AlertManager:
    def __init__(self):
        self.alerts = []
        self.last_triggered = {}

    def add_alert(self, alert: Alert):
        self.alerts.append(alert)

    def check_alerts(self, metrics: dict):
        for alert in self.alerts:
            if alert.condition(metrics):
                # Check cooldown
                if alert.name in self.last_triggered:
                    elapsed = time.time() - self.last_triggered[alert.name]
                    if elapsed < alert.cooldown:
                        continue

                # Trigger alert
                alert.action(metrics)
                self.last_triggered[alert.name] = time.time()

# Define alerts
alerts = AlertManager()

alerts.add_alert(Alert(
    name="low_success_rate",
    condition=lambda m: m['success_rate'] < 0.90,
    action=lambda m: send_slack_alert(f"Success rate dropped to {m['
    success_rate']:.2%}")
))

alerts.add_alert(Alert(
    name="high_latency",
```

```
    condition=lambda m: m['p95_latency'] > 5.0,
    action=lambda m: send_pagerduty(f"P95 latency: {m['p95_latency']:.1f}s")
))

# Run checks every minute
while True:
    metrics = collect_metrics()
    alerts.check_alerts(metrics)
    time.sleep(60)
```

1.1.7 Part 6: Advanced Error Recovery

Circuit Breaker Pattern

```
from enum import Enum
import time

class CircuitState(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"

class CircuitBreaker:
    def __init__(self, failure_threshold: int = 5, timeout: int = 60):
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.failures = 0
        self.last_failure_time = None
        self.state = CircuitState.CLOSED

    def call(self, func, *args, **kwargs):
        if self.state == CircuitState.OPEN:
            if time.time() - self.last_failure_time > self.timeout:
                self.state = CircuitState.HALF_OPEN
            else:
                raise Exception("Circuit breaker is OPEN")

        try:
            result = func(*args, **kwargs)
            self._on_success()
            return result

        except Exception as e:
            self._on_failure()
            raise

    def _on_success(self):
        self.failures = 0
        if self.state == CircuitState.HALF_OPEN:
            self.state = CircuitState.CLOSED

    def _on_failure(self):
        self.failures += 1
        self.last_failure_time = time.time()

        if self.failures >= self.failure_threshold:
            self.state = CircuitState.OPEN

# Use it
breaker = CircuitBreaker(failure_threshold=5, timeout=60)

def extract_with_circuit_breaker(text: str) -> dict:
    return breaker.call(expensive_api_call, text)
```

1.1.8 Part 7: Cost Optimization

Smart Model Selection

```
def select_model(text: str, complexity: str = "auto") -> str:
    """Choose cheapest model that meets requirements"""

    if complexity == "auto":
        complexity = estimate_complexity(text)

    if complexity == "simple":
        return "gpt-3.5-turbo"  # $0.001/1K tokens
    elif complexity == "medium":
        return "gpt-4"          # $0.03/1K tokens
    else:
        return "gpt-4-32k"      # $0.06/1K tokens

def estimate_complexity(text: str) -> str:
    """Estimate task complexity"""

    # Simple heuristics
    if len(text) < 500:
        return "simple"
    elif len(text) < 2000:
        return "medium"
    else:
        return "complex"
```

Cost Tracking

```
class CostTracker:
    def __init__(self):
        self.total_cost = 0
        self.costs_by_model = {}

    def log_request(self, model: str, input_tokens: int, output_tokens: int):
        """Track costs per request"""

        # Pricing (example)
        prices = {
            "gpt-4": {"input": 0.03, "output": 0.06},
            "gpt-3.5-turbo": {"input": 0.001, "output": 0.002}
        }

        cost = (
            (input_tokens / 1000) * prices[model]["input"] +
            (output_tokens / 1000) * prices[model]["output"]
        )

        self.total_cost += cost

        if model not in self.costs_by_model:
            self.costs_by_model[model] = 0
        self.costs_by_model[model] += cost

    def get_report(self) -> dict:
        return {
            "total": self.total_cost,
            "by_model": self.costs_by_model,
            "avg_per_request": self.total_cost / sum(self.costs_by_model.
    values())
        }
```

1.1.9 Part 8: Security Considerations

Input Sanitization

```python
import re

def sanitize_input(text: str) -> str:
    """Remove potentially malicious content"""

    # Remove potential prompt injection attempts
    dangerous_patterns = [
        r"ignore previous instructions",
        r"system:",
        r"<\|.*?\|>",
        r"###\s*instruction",
    ]

    for pattern in dangerous_patterns:
        text = re.sub(pattern, "[removed]", text, flags=re.IGNORECASE)

    # Limit length
    max_length = 10000
    if len(text) > max_length:
        text = text[:max_length]

    return text
```

Output Validation

```python
def validate_output_safety(data: dict) -> bool:
    """Ensure output doesn't contain sensitive data"""

    # Check for PII patterns
    pii_patterns = [
        r"\b\d{3}-\d{2}-\d{4}\b",  # SSN
        r"\b\d{16}\b",              # Credit card
        r"\b[\w.-]+@[\w.-]+\.\w+\b" # Email
    ]

    data_str = json.dumps(data)

    for pattern in pii_patterns:
        if re.search(pattern, data_str):
            logger.warning("PII detected in output")
            return False

    return True
```

1.1.10 Key Takeaways

1. **Validation Pipelines**: Multi-stage validation catches more errors
2. **Multi-Model Fallback**: Don't rely on single model
3. **Aggressive Caching**: 60-80% cost reduction possible
4. **Comprehensive Monitoring**: Track everything in production
5. **Circuit Breakers**: Prevent cascade failures
6. **Cost Optimization**: Choose cheapest model for each task
7. **Security First**: Sanitize inputs, validate outputs
8. **Alert Fatigue**: Set meaningful thresholds

1.1.11 Production Readiness Checklist

- ☐ Multi-stage validation implemented
- ☐ At least 2 fallback strategies
- ☐ Caching layer (Redis)
- ☐ Comprehensive monitoring (Prometheus/Datadog)
- ☐ Alerting configured
- ☐ Circuit breakers on external APIs
- ☐ Cost tracking and budgets
- ☐ Security review completed
- ☐ Load tested at 10x expected volume
- ☐ Runbook created
- ☐ Team trained on system
- ☐ Rollback plan documented

You're now ready to build enterprise-grade structured AI systems!