Generative AI - Intermediate Handout

Machine Learning for Smarter Innovation

1 Generative AI - 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 hashlib
import time
from typing import List, Dict, Optional
from dataclasses import dataclass
import pandas as pd
import numpy as np

# API clients
from anthropic import Anthropic
from openai import OpenAI

# For structured outputs
from pydantic import BaseModel, Field

# Environment setup
import warnings
warnings.filterwarnings('ignore')

# API initialization (use environment variables in production)
# anthropic_client = Anthropic(api_key=os.environ.get("ANTHROPIC_API_KEY"))
# openai_client = OpenAI(api_key=os.environ.get("OPENAI_API_KEY"))
```

This handout covers practical implementation of generative AI systems for design and innovation tasks. Generative AI transforms text prompts into creative outputs spanning concepts, copy, code, and structured data. The techniques apply across product ideation, marketing automation, content generation, and design system creation. We focus on prompt engineering patterns that produce consistent, high-quality results for business applications.

1.2 1. Prompt Engineering Fundamentals

1.2.1 Concept Overview

Prompt engineering is the practice of crafting inputs that guide language models toward desired outputs. Effective prompts combine clear instructions, relevant context, output format specifications, and examples. The quality difference between a naive prompt and an engineered prompt often exceeds the difference between model generations.

1.2.2 Implementation: Core Prompting Patterns

```
class PromptBuilder:
    """Structured prompt construction for consistent results."""

    def __init__(self, model_name: str = "claude-sonnet-4-20250514"):
        self.model = model_name
        self.components = {
            'role': '',
            'context': '',
            'task': '',
            'constraints': [],
            'output_format': '',
            'examples': []
        }

    def set_role(self, role: str) -> 'PromptBuilder':
        """Set the persona for the AI to adopt."""
        self.components['role'] = role
        return self

    def set_context(self, context: str) -> 'PromptBuilder':
        """Provide background information."""
        self.components['context'] = context
        return self

    def set_task(self, task: str) -> 'PromptBuilder':
        """Define the specific task to accomplish."""
        self.components['task'] = task
        return self

    def add_constraint(self, constraint: str) -> 'PromptBuilder':
        """Add a constraint or requirement."""
        self.components['constraints'].append(constraint)
        return self

    def set_output_format(self, format_spec: str) -> 'PromptBuilder':
        """Specify the desired output format."""
        self.components['output_format'] = format_spec
        return self

    def add_example(self, input_text: str, output_text: str) -> 'PromptBuilder
    ':
        """Add a few-shot learning example."""
        self.components['examples'].append({
            'input': input_text,
            'output': output_text
        })
        return self

    def build(self) -> str:
        """Construct the final prompt."""
```

```
        parts = []

        if self.components['role']:
            parts.append(f"You are {self.components['role']}.")

        if self.components['context']:
            parts.append(f"\nContext: {self.components['context']}")

        if self.components['examples']:
            parts.append("\nExamples:")
            for ex in self.components['examples']:
                parts.append(f"Input: {ex['input']}")
                parts.append(f"Output: {ex['output']}\n")

        if self.components['task']:
            parts.append(f"\nTask: {self.components['task']}")

        if self.components['constraints']:
            parts.append("\nConstraints:")
            for c in self.components['constraints']:
                parts.append(f"- {c}")

        if self.components['output_format']:
            parts.append(f"\nOutput Format: {self.components['output_format']}
    ")

        return '\n'.join(parts)


# Usage example
builder = PromptBuilder()
prompt = (builder
    .set_role("a senior product designer at a leading tech company")
    .set_context("We are developing a mobile app for small business invoicing"
    )
    .add_example(
        "User needs fitness tracking",
        "FitTrack: Minimalist app with daily goals, progress rings, social
    challenges"
    )
    .add_example(
        "User needs recipe management",
        "ChefMate: Smart organizer with meal planning, shopping lists,
    nutrition tracking"
    )
    .set_task("Design a compelling product concept for the invoicing app")
    .add_constraint("Must work offline for field workers")
    .add_constraint("Maximum 3-tap workflow for common actions")
    .add_constraint("Accessible to users with limited tech experience")
    .set_output_format("Product name, tagline, 3 key features, and target user
     persona")
    .build())

print(prompt)
```

1.3 2. Chain-of-Thought and Reasoning Prompts

1.3.1 Concept Overview

Chain-of-thought prompting guides models through explicit reasoning steps, improving performance on complex tasks. By asking models to show their work, we get more reliable outputs and can identify where reasoning goes wrong. This technique is essential for design decisions requiring multiple considerations.

1.3.2 Implementation: Structured Reasoning

```
@dataclass
class ReasoningStep:
    """Represents one step in a reasoning chain."""
    step_number: int
    description: str
    prompt_text: str
    expected_output_type: str


class ChainOfThoughtPrompt:
    """Build multi-step reasoning prompts."""

    def __init__(self, problem_statement: str):
        self.problem = problem_statement
        self.steps: List[ReasoningStep] = []

    def add_step(self, description: str, prompt_text: str,
                 output_type: str = "analysis") -> 'ChainOfThoughtPrompt':
        """Add a reasoning step to the chain."""
        step = ReasoningStep(
            step_number=len(self.steps) + 1,
            description=description,
            prompt_text=prompt_text,
            expected_output_type=output_type
        )
        self.steps.append(step)
        return self

    def build(self) -> str:
        """Construct the chain-of-thought prompt."""
        parts = [f"Problem: {self.problem}\n"]
        parts.append("Let's solve this step by step:\n")

        for step in self.steps:
            parts.append(f"Step {step.step_number}: {step.description}")
            parts.append(f"   {step.prompt_text}")
            parts.append(f"   Expected output: {step.expected_output_type}\n")

        parts.append("Show your reasoning for each step before providing the
    final answer.")
        return '\n'.join(parts)


# Design problem example
cot_prompt = ChainOfThoughtPrompt(
    "Design a sustainable packaging solution for an e-commerce company"
)

cot_prompt.add_step(
    "Problem Analysis",
    "Identify the key environmental and practical problems with current e-
    commerce packaging",
```

```
    "List of 3-5 specific problems with evidence"
).add_step(
    "Material Research",
    "List available eco-friendly materials and their properties",
    "Comparison table of materials with pros/cons"
).add_step(
    "User Requirements",
    "Consider what customers need from packaging (protection, unboxing
    experience, disposal)",
    "Prioritized list of user requirements"
).add_step(
    "Constraint Analysis",
    "Identify cost, manufacturing, and logistics constraints",
    "Feasibility assessment"
).add_step(
    "Solution Synthesis",
    "Propose 3 innovative solutions that balance all factors",
    "Detailed solution descriptions with tradeoffs"
)

print(cot_prompt.build())
```

1.4 3. API Integration and Production Patterns

1.4.1 Concept Overview

Moving from playground experimentation to production requires robust API integration with error handling, rate limiting, cost tracking, and caching. Production systems must handle failures gracefully and optimize for both quality and cost.

1.4.2 Implementation: Production-Ready API Client

```
class GenerativeAIClient:
    """Production-ready client for generative AI APIs."""

    def __init__(self, provider: str = "anthropic"):
        self.provider = provider
        self.cache: Dict[str, str] = {}
        self.usage_log: List[Dict] = []
        self.retry_delays = [1, 2, 4, 8, 16]  # Exponential backoff

        # Cost per 1M tokens (approximate, verify current pricing)
        self.costs = {
            'anthropic': {'input': 3.0, 'output': 15.0},
            'openai_gpt4': {'input': 10.0, 'output': 30.0},
            'openai_gpt35': {'input': 0.5, 'output': 1.5}
        }

    def _get_cache_key(self, prompt: str, params: Dict) -> str:
        """Generate cache key from prompt and parameters."""
        cache_input = json.dumps({'prompt': prompt, 'params': params},
    sort_keys=True)
        return hashlib.sha256(cache_input.encode()).hexdigest()

    def _log_usage(self, prompt: str, response: str,
                   input_tokens: int, output_tokens: int,
                   latency: float, cached: bool):
```

```
        """Track API usage for monitoring and cost analysis."""
        cost_rates = self.costs.get(self.provider, self.costs['anthropic'])
        cost = (input_tokens * cost_rates['input'] +
                output_tokens * cost_rates['output']) / 1_000_000

        self.usage_log.append({
            'timestamp': time.time(),
            'provider': self.provider,
            'input_tokens': input_tokens,
            'output_tokens': output_tokens,
            'cost_usd': cost if not cached else 0,
            'latency_seconds': latency,
            'cached': cached
        })

    def generate(self, prompt: str,
                 temperature: float = 0.7,
                 max_tokens: int = 1000,
                 use_cache: bool = True) -> str:
        """Generate response with caching and error handling."""

        params = {'temperature': temperature, 'max_tokens': max_tokens}
        cache_key = self._get_cache_key(prompt, params)

        # Check cache
        if use_cache and cache_key in self.cache:
            self._log_usage(prompt, self.cache[cache_key],
                            len(prompt.split()), len(self.cache[cache_key].split
    ()),
                            0, cached=True)
            return self.cache[cache_key]

        # Simulated API call (replace with actual API call)
        start_time = time.time()
        response = self._call_api(prompt, params)
        latency = time.time() - start_time

        # Cache and log
        self.cache[cache_key] = response
        self._log_usage(prompt, response,
                        len(prompt.split()) * 1.3,  # Approximate tokens
                        len(response.split()) * 1.3,
                        latency, cached=False)

        return response

    def _call_api(self, prompt: str, params: Dict) -> str:
        """Make API call with retry logic."""
        # This is a simulation - replace with actual API calls
        # For real implementation:
        # if self.provider == 'anthropic':
        #     response = anthropic_client.messages.create(...)
        # elif self.provider == 'openai':
        #     response = openai_client.chat.completions.create(...)

        return f"[Simulated response for: {prompt[:50]}...]"

    def get_usage_summary(self) -> pd.DataFrame:
        """Get summary of API usage and costs."""
        if not self.usage_log:
            return pd.DataFrame()

        df = pd.DataFrame(self.usage_log)
```

```
        summary = {
            'total_calls': len(df),
            'cached_calls': df['cached'].sum(),
            'total_cost_usd': df['cost_usd'].sum(),
            'avg_latency_seconds': df['latency_seconds'].mean(),
            'total_input_tokens': df['input_tokens'].sum(),
            'total_output_tokens': df['output_tokens'].sum()
        }
        return pd.DataFrame([summary])


# Usage example
client = GenerativeAIClient(provider='anthropic')

# First call - hits API
response1 = client.generate(
    "Design a mobile app for plant care tracking",
    temperature=0.7,
    max_tokens=500
)

# Second identical call - returns cached response
response2 = client.generate(
    "Design a mobile app for plant care tracking",
    temperature=0.7,
    max_tokens=500
)

print(client.get_usage_summary())
```

1.5 4. Structured Output Generation

1.5.1 Concept Overview

Business applications often require structured outputs like JSON, tables, or specific formats. Prompt engineering combined with output parsing ensures consistent, machine-readable results. Using schema definitions and validation catches format errors before they propagate through systems.

1.5.2 Implementation: Schema-Validated Outputs

```
class ProductConcept(BaseModel):
    """Schema for product concept outputs."""
    name: str = Field(description="Product name")
    tagline: str = Field(max_length=100, description="Marketing tagline")
    problem_solved: str = Field(description="Core problem addressed")
    target_user: str = Field(description="Primary user persona")
    key_features: List[str] = Field(min_length=3, max_length=5)
    competitive_advantage: str = Field(description="Key differentiator")
    monetization: str = Field(description="Revenue model")
    estimated_complexity: str = Field(description="low/medium/high")


class StructuredOutputGenerator:
    """Generate and validate structured outputs from LLMs."""

    def __init__(self, client: GenerativeAIClient):
        self.client = client
```

```
    def generate_product_concepts(self,
                                  brief: str,
                                  num_concepts: int = 3) -> List[Dict]:
        """Generate validated product concepts from a brief."""

        schema_description = """
        Return a JSON array with the following structure for each concept:
        {
            "name": "Product Name",
            "tagline": "Short marketing tagline (max 100 chars)",
            "problem_solved": "Core problem this addresses",
            "target_user": "Primary user persona description",
            "key_features": ["Feature 1", "Feature 2", "Feature 3"],
            "competitive_advantage": "What makes this unique",
            "monetization": "Revenue model",
            "estimated_complexity": "low|medium|high"
        }
        """

        prompt = f"""
        Create {num_concepts} distinct product concepts based on this brief:

        Brief: {brief}

        Requirements:
        - Each concept must solve the problem differently
        - Be specific and actionable, not generic
        - Consider technical feasibility
        - Include realistic monetization strategies

        Output format:
        {schema_description}

        Return ONLY valid JSON, no additional text.
        """

        response = self.client.generate(prompt, temperature=0.8, max_tokens
    =2000)

        # Parse and validate
        try:
            concepts = json.loads(response)
            validated = []
            for c in concepts:
                try:
                    concept = ProductConcept(**c)
                    validated.append(concept.model_dump())
                except Exception as e:
                    print(f"Validation error: {e}")
            return validated
        except json.JSONDecodeError:
            # Attempt to extract JSON from response
            return self._extract_json(response)

    def _extract_json(self, text: str) -> List[Dict]:
        """Extract JSON from text that may contain other content."""
        import re
        # Find JSON array pattern
        match = re.search(r'\[[\s\S]*\]', text)
        if match:
            try:
                return json.loads(match.group())
```

```
            except json.JSONDecodeError:
                pass
        return []


# Usage
client = GenerativeAIClient()
generator = StructuredOutputGenerator(client)

concepts = generator.generate_product_concepts(
    brief="A solution for remote teams struggling with asynchronous
    communication",
    num_concepts=3
)

for i, concept in enumerate(concepts, 1):
    print(f"\nConcept {i}:")
    print(json.dumps(concept, indent=2))
```

1.6 5. Multi-Turn Conversation Patterns

1.6.1 Concept Overview

Complex design tasks benefit from iterative refinement through multi-turn conversations. Managing conversation history, providing feedback, and building on previous responses enables deeper exploration of solution spaces. Effective conversation design prevents context drift while allowing creative expansion.

1.6.2 Implementation: Conversation Manager

```
@dataclass
class Message:
    """Single message in a conversation."""
    role: str  # 'user', 'assistant', or 'system'
    content: str
    timestamp: float = None

    def __post_init__(self):
        if self.timestamp is None:
            self.timestamp = time.time()


class ConversationManager:
    """Manage multi-turn design conversations."""

    def __init__(self, client: GenerativeAIClient, system_prompt: str = None):
        self.client = client
        self.history: List[Message] = []

        if system_prompt:
            self.history.append(Message(role='system', content=system_prompt))

    def add_user_message(self, content: str):
        """Add user message to history."""
        self.history.append(Message(role='user', content=content))

    def generate_response(self,
                          include_history: bool = True,
```

```
                        max_history_turns: int = 10) -> str:
        """Generate assistant response based on conversation history."""

        # Build context from history
        if include_history:
            recent_history = self.history[-max_history_turns * 2:]
            context_parts = []
            for msg in recent_history:
                context_parts.append(f"{msg.role.upper()}: {msg.content}")
            context = "\n\n".join(context_parts)
        else:
            context = self.history[-1].content if self.history else ""

        prompt = f"""
        Conversation history:
        {context}

        Continue this conversation as the assistant. Build on previous
    messages,
        maintain consistency with earlier decisions, and provide helpful,
    specific responses.
        """

        response = self.client.generate(prompt, temperature=0.7)
        self.history.append(Message(role='assistant', content=response))

        return response

    def refine_concept(self, feedback: str) -> str:
        """Provide feedback and get refined output."""
        refinement_prompt = f"""
        Based on this feedback: {feedback}

        Revise the previous response to address these concerns while
    maintaining
        the strengths of the original concept.
        """
        self.add_user_message(refinement_prompt)
        return self.generate_response()

    def branch_conversation(self, branch_point: int = None) -> '
    ConversationManager':
        """Create a branch of the conversation for exploring alternatives."""
        new_manager = ConversationManager(self.client)
        if branch_point:
            new_manager.history = self.history[:branch_point].copy()
        else:
            new_manager.history = self.history.copy()
        return new_manager

    def get_summary(self) -> str:
        """Get a summary of the conversation."""
        if len(self.history) < 3:
            return "Conversation too short to summarize"

        history_text = "\n".join([f"{m.role}: {m.content[:200]}..."
                                  for m in self.history])

        summary_prompt = f"""
        Summarize this design conversation in 3-4 bullet points,
        highlighting key decisions and current direction:

        {history_text}
```

```
        """

        return self.client.generate(summary_prompt, max_tokens=300)


# Usage example
client = GenerativeAIClient()

system_prompt = """You are a senior product designer specializing in mobile
    applications.
You provide specific, actionable design recommendations backed by user
    research principles.
You ask clarifying questions when requirements are ambiguous."""

conversation = ConversationManager(client, system_prompt)

# Initial request
conversation.add_user_message(
    "I need to design an app for elderly users to manage their medications"
)
response1 = conversation.generate_response()
print("Initial response:", response1)

# Refinement
response2 = conversation.refine_concept(
    "Good start, but we need larger touch targets and voice control options"
)
print("Refined response:", response2)

# Branch for alternative exploration
alternative = conversation.branch_conversation(branch_point=2)
alternative.add_user_message("What if we focused on caregiver features instead
    ?")
alt_response = alternative.generate_response()
print("Alternative direction:", alt_response)
```

1.7 6. Batch Processing and Evaluation

1.7.1 Concept Overview

Generating multiple variations and systematically evaluating them helps identify the best outputs. Batch processing enables comparing different prompts, temperatures, and models. Automated evaluation with rubrics provides consistent quality assessment across many generations.

1.7.2 Implementation: Batch Generation and Evaluation

```
@dataclass
class EvaluationCriteria:
    """Criteria for evaluating generated content."""
    name: str
    description: str
    weight: float = 1.0


class BatchGenerator:
    """Generate and evaluate multiple variations."""
```

```
    def __init__(self, client: GenerativeAIClient):
        self.client = client
        self.results: List[Dict] = []

    def generate_variations(self,
                            base_prompt: str,
                            variations: List[Dict],
                            num_samples: int = 3) -> List[Dict]:
        """Generate multiple variations with different parameters."""

        all_results = []

        for var in variations:
            var_name = var.get('name', 'default')
            temperature = var.get('temperature', 0.7)
            prompt_modifier = var.get('prompt_modifier', '')

            full_prompt = f"{base_prompt}\n\n{prompt_modifier}" if
    prompt_modifier else base_prompt

            for i in range(num_samples):
                response = self.client.generate(
                    full_prompt,
                    temperature=temperature,
                    use_cache=False  # Want unique samples
                )

                all_results.append({
                    'variation': var_name,
                    'sample_id': i + 1,
                    'temperature': temperature,
                    'prompt': full_prompt,
                    'response': response
                })

        self.results = all_results
        return all_results

    def evaluate_results(self,
                         criteria: List[EvaluationCriteria]) -> pd.DataFrame:
        """Evaluate generated results against criteria using LLM."""

        evaluation_results = []

        for result in self.results:
            criteria_text = "\n".join([
                f"- {c.name}: {c.description}" for c in criteria
            ])

            eval_prompt = f"""
            Evaluate this generated content on a scale of 1-5 for each
    criterion:

            Content:
            {result['response']}

            Criteria:
            {criteria_text}

            Return scores as JSON: {{"criterion_name": score, ...}}
            Also include "reasoning" with brief explanations.
            """
```

```
            eval_response = self.client.generate(eval_prompt, temperature=0.3)

            try:
                scores = json.loads(eval_response)
            except json.JSONDecodeError:
                scores = {c.name: 3 for c in criteria}  # Default middle score

            # Calculate weighted score
            weighted_sum = sum(
                scores.get(c.name, 3) * c.weight for c in criteria
            )
            total_weight = sum(c.weight for c in criteria)

            evaluation_results.append({
                **result,
                'scores': scores,
                'weighted_score': weighted_sum / total_weight
            })

        return pd.DataFrame(evaluation_results)

    def get_best_results(self, df: pd.DataFrame, top_n: int = 3) -> pd.
    DataFrame:
        """Return top N results by weighted score."""
        return df.nlargest(top_n, 'weighted_score')


# Usage
client = GenerativeAIClient()
batch = BatchGenerator(client)

# Define variations to test
variations = [
    {'name': 'creative', 'temperature': 0.9, 'prompt_modifier': 'Be highly
    creative and unconventional.'},
    {'name': 'balanced', 'temperature': 0.7, 'prompt_modifier': ''},
    {'name': 'conservative', 'temperature': 0.3, 'prompt_modifier': 'Focus on
    proven, practical solutions.'}
]

# Generate variations
results = batch.generate_variations(
    base_prompt="Design a novel approach to urban bicycle parking",
    variations=variations,
    num_samples=2
)

# Define evaluation criteria
criteria = [
    EvaluationCriteria('originality', 'How novel and unique is the idea?',
    weight=1.5),
    EvaluationCriteria('feasibility', 'How practical to implement?', weight
    =1.0),
    EvaluationCriteria('user_value', 'How much value for end users?', weight
    =1.2),
    EvaluationCriteria('clarity', 'How clear and well-articulated?', weight
    =0.8)
]

# Evaluate
evaluated_df = batch.evaluate_results(criteria)
best = batch.get_best_results(evaluated_df, top_n=3)
print(best[['variation', 'weighted_score', 'response']])
```

1.8 Common Parameters and Settings

()	()	()	()
*	*	*	*
0.2500	[illegible]	[illegible]	Typical use 0.3409 Range Case

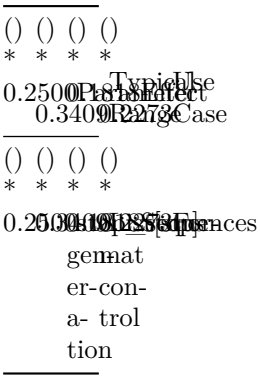
()	()	()	()
*	*	*	*
0.2500	[illegible]	[illegible]	[illegible] 0.3 per- a- ture 1.0 con- trol ran- dom- ness 0.9 for cre- ativ- ity

()	()	()	()
*	*	*	*
0.2500	[illegible]	[illegible]	[illegible] Match - put to length ex- 4000 pected re- sponse length

()	()	()	()
*	*	*	*
0.2500	[illegible]	[illegible]	[illegible] Al- - cle- ter sam- na- 1.0 plin- tive ity to tem- per- a- ture

()	()	()	()
*	*	*	*
0.2500	[illegible]	[illegible]	[illegible] 0.3 - ence 2.0 pres- alty - ages for rep- va- 2.0 e- ri- ti- ety tion

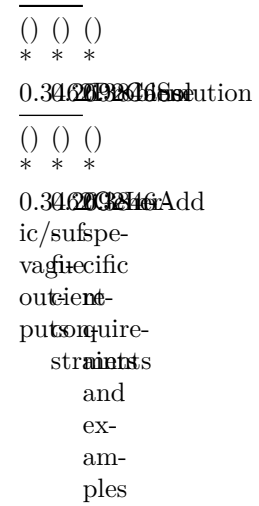
()	()	()	()
*	*	*	*
0.2500	[illegible]	[illegible]	[illegible] 0.3 - quency 2.0 fre- alty - word for rep- nat- 2.0 e- u- ti- ral tion next



1.9 Practice Projects

1. **Design System Generator**: Build a system that generates complete design systems (colors, typography, components) from a brand brief. Include validation that ensures accessibility compliance and consistency across components.
2. **Competitive Analysis Automation**: Create a pipeline that takes competitor product descriptions and generates comparative analysis including feature matrices, positioning maps, and strategic recommendations.
3. **User Research Synthesizer**: Develop a tool that processes user interview transcripts and generates persona documents, journey maps, and key insights with supporting quotes.
4. **A/B Test Content Generator**: Build a system that generates multiple headline and copy variations for A/B testing, with automated evaluation scoring for predicted click-through rates.

1.10 Troubleshooting



() *	() *	() *
0.3	4	Resolution
() *	() *	() *
0.3	4	Use consistent terminology for similar concepts
0.3	4	Log LLM costs by model and feature
0.3	4	Ground answers in provided context
0.3	4	Prefer bullet lists over long prose
() *	() *	() *
0.3	4	Use explicit JSON schemas with validation
() *	() *	() *
0.3	4	Log LLM costs by model and feature
() *	() *	() *
0.3	4	Ground answers in provided context
() *	() *	() *
0.3	4	Prefer bullet lists over long prose

() *	() *	() *
0.3	4	2
() *	() *	() *
0.3	4	2
limiter-rapid-rose-quest	implement-exponential-backoff-and-queuing	4
() *	() *	() *
0.3	4	2
cated-outputs	decrease-low-limit-or-request-in-chunks	max_tokens
() *	() *	() *
0.3	4	2
input-validation-vulnerabilities	sanitize-validated-user-input-system-prompts	4

1.11 Next Steps

- Read the advanced handout for fine-tuning, RAG systems, and agent architectures
- Experiment with multi-modal models (text + image generation)
- Implement evaluation pipelines with human feedback collection
- Explore vector databases for retrieval-augmented generation
- Build production monitoring dashboards for AI system performance

---

Generative AI amplifies human creativity when guided by well-crafted prompts. The gap between amateur and expert prompt engineering often exceeds the gap between model generations. Invest in systematic prompt development, evaluation, and iteration to maximize the value of these powerful tools.