



GenAI & Prompting

Stage 3: Large Language Models with Few-Shot Learning

The breakthrough that changed everything — LLMs that understand natural language, learn from examples, and generate human-like responses. Fast to build, but requires careful prompt engineering.



What is GenAI + Prompting?

TL;DR

Use Large Language Models (like Gemini, GPT-4, Claude) with carefully crafted prompts to solve problems. No training data needed — just write instructions and provide examples. The model figures out the pattern and applies it to new cases. Fast to prototype but can hallucinate without proper constraints.

The Revolutionary Change

✗ Before GenAI (Stages 1-2)

- **Rule-Based:** Write explicit rules for every case
- **Traditional ML:** Collect 1000s of labeled examples
- **Feature Engineering:** Manually craft features
- **Training Time:** Days to weeks
- **Rigid:** Breaks with new patterns

Development: 6-8 weeks

With GenAI (Stage 3)

- **Natural Language:** Write instructions in plain English
- **Few-Shot Learning:** Provide 3-5 examples
- **Zero Training:** No model training needed
- **Immediate:** Works instantly
- **Flexible:** Adapts to new patterns

 **Development: 1-2 days**

How It Works for L1 IT Support

Instead of writing code or training models, you write a prompt:

```
[ROLE] You are an L1 IT Support Agent for a banking data platform. [CONTEXT]
You handle issues like: pipeline failures, data quality, access requests, query
optimization. [EXAMPLES] Ticket: "Pipeline job-456 failed with schema mismatch
error" Response: This is a schema validation issue. Run schema_sync_job for the
affected table, then restart the pipeline. ETA: 10 minutes. Ticket: "Query
timeout on dashboard_report" Response: Your query exceeded the 30s limit. Add
indexes on frequently filtered columns or increase the timeout setting. I can
help with either approach. [TASK] New Ticket: {user_ticket} Provide: 1. Issue
category 2. Root cause analysis 3. Step-by-step solution 4. Estimated
resolution time
```

That's it! The LLM understands the pattern from examples and applies it to new tickets.
No training, no feature engineering, no weeks of development.



Architecture & Implementation

Basic Setup with Gemini

```
# Install Gemini SDK pip install google-generativeai --break-system-packages # Set up authentication export GOOGLE_API_KEY="your-api-key-here"
```

Simple Implementation

```
import google.generativeai as genai from typing import Dict # Configure API genai.configure(api_key="YOUR_API_KEY") # Initialize model model = genai.GenerativeModel('gemini-1.5-pro') def classify_support_ticket(ticket_description: str) -> Dict: """ Classify and provide solution for support ticket using GenAI. Args: ticket_description: User's ticket description Returns: Dict with category, solution, and reasoning """ # Construct prompt with role, examples, and task prompt = f"""You are an L1 IT Support Agent for a banking data platform. ROLE: Analyze support tickets and provide solutions for: - Pipeline failures (schema errors, timeouts, dependencies) - Data quality issues (null values, duplicates, freshness) - Access requests (permissions, API keys) - Query optimization (slow queries, resource usage) EXAMPLES: Ticket: "Pipeline job-123 failed with schema mismatch error in users table" Analysis: - Category: schema_validation_error - Root Cause: Column definition mismatch between source and target - Solution: Run schema_sync_job for users table, then restart pipeline - Time: 10 minutes - Confidence: High Ticket: "Dashboard query timing out after 30 seconds" Analysis: - Category: query_optimization - Root Cause: Query exceeding timeout limit, likely missing indexes - Solution: Add indexes on date_column and user_id, or increase timeout to 60s - Time: 20 minutes - Confidence: High Ticket: "Need access to prod_analytics database for reporting" Analysis: - Category: access_request - Root Cause: User lacks read permissions - Solution: Grant viewer role after manager approval verification - Time: 5 minutes - Confidence: High NEW TICKET: {ticket_description} Provide your analysis in the same format: - Category: [category_name] - Root Cause: [brief explanation] - Solution: [step-by-step instructions] - Time: [estimated resolution time] - Confidence: [High/Medium/Low] """ # Generate response response = model.generate_content(prompt) # Parse response (in production, use structured output) return { "raw_response": response.text, "model_used": "gemini-1.5-pro" } # Example usage ticket = "Getting null values in customer_transactions.amount field" result = classify_support_ticket(ticket) print(result["raw_response"])
```

Performance Metrics for L1 Support

Metric	Value	Notes
Ticket Coverage	~50%	Similar to ML but without training
Response Time	2-5 seconds	Depends on prompt length and model
Development Time	1-2 weeks	Mostly prompt engineering and testing
Cost (500 tickets/month)	\$2,000-\$5,000	Based on Gemini Pro pricing
Hallucination Rate	~30%	Without constraints — this is the problem!
Consistency	Medium	Same prompt can give different outputs

Prompt Engineering Techniques

1. Zero-Shot Prompting

What It Is

Give the model a task with no examples. Relies purely on the model's pre-trained knowledge.

When to Use

- Simple, well-known tasks
- General knowledge questions
- Quick prototyping

Example

```
Classify this support ticket into one of these categories: -
schema_error - timeout - access_request - data_quality Ticket: "Pipeline
failed with schema mismatch" Category:
```

Result

Model Output: schema_error

Limitation: Model might not understand your specific categories or domain terminology. Success rate: ~40-60% for domain-specific tasks.

2. Few-Shot Prompting (Recommended)

What It Is

Provide 3-5 examples showing the pattern you want. Model learns from examples and applies to new cases.

When to Use

- Domain-specific tasks
- Custom output formats
- When you need consistent behavior

Example

```
Classify support tickets using these examples: Example 1: Ticket: "Job timed out after 30 minutes" Category: pipeline_timeout Severity: high Solution: Increase timeout setting Example 2: Ticket: "Need viewer access to analytics DB" Category: access_request Severity: low Solution: Grant read-only role Example 3: Ticket: "Null values in amount column" Category: data_quality Severity: high Solution: Apply data validation rules Now classify: Ticket: "Schema validation failed on users table" Respond in the same format.
```

Result

Model Output:

Category: schema_validation_error
Severity: high
Solution: Run schema sync job and restart pipeline

Success Rate: ~70-80% for domain-specific tasks with good examples.

3. Chain-of-Thought (CoT) Prompting

What It Is

Ask the model to show its reasoning step-by-step before giving the final answer.

When to Use

- Complex problem-solving
- Multi-step reasoning
- When you need to verify logic

Example

Analyze this support ticket step-by-step: Ticket: "Dashboard showing stale data from 3 days ago, but pipeline ran successfully" Think through this systematically: 1. What could cause stale data despite successful pipeline? 2. What components should we check? 3. What's the most likely root cause? 4. What's the solution? Show your reasoning, then provide the final answer.

Result

Model Output:

Step 1 - Possible Causes:

- Pipeline might be reading from wrong source
- Dashboard cache not refreshing
- Data refresh job running but not updating target table

Step 2 - Components to Check:

- Pipeline source configuration
- Dashboard data source connection
- Last update timestamp in target table

Step 3 - Most Likely:

Dashboard is connected to old data source or has aggressive caching

Step 4 - Solution:

Check dashboard data source, clear cache, verify it points to correct table

Benefit: Improved accuracy on complex tasks (20-30% better) and explainable reasoning.

4. Role Prompting

What It Is

Assign the model a specific role/persona to guide its behavior and tone.

Example

You are a senior L1 support engineer with 5 years experience in banking data platforms. You are known for: - Clear, concise explanations - Systematic troubleshooting approach - Always including specific SOP references - Empathetic communication with users When handling tickets, you always: 1. Acknowledge the user's issue 2. Explain what you found 3. Provide step-by-step solution 4. Give realistic time estimates 5. Sign off professionally Now handle this ticket: "Pipeline keeps failing, please help!"

Result

Model Output:

I understand you're experiencing pipeline failures — let me help you resolve this.

What I Found:

Based on the error pattern, this appears to be a recurring timeout issue...

Solution:

1. First, check current timeout setting (SOP-234)
2. Review job execution logs for bottlenecks
3. Increase timeout to 2x current value
4. Test with a small batch

Timeline: This should take about 20 minutes to implement and verify.

Let me know if you need any clarification!
— Support Team

5. Structured Output Prompting

What It Is

Force the model to output in a specific format (JSON, XML, table, etc.).

Example

```
Analyze this ticket and respond ONLY with valid JSON: { "ticket_id":  
  "string", "category": "string", "severity": "high|medium|low",  
  "root_cause": "string", "solution_steps": ["step1", "step2", ...],  
  "estimated_time_minutes": integer, "requires_approval": boolean,  
  "sop_reference": "string or null" } Ticket: "Need emergency access to  
  production database for audit" Output ONLY the JSON, nothing else:
```

Result

```
{ "ticket_id": "TKT-AUDIT-001", "category": "access_request_emergency",  
  "severity": "high", "root_cause": "Audit team needs read access for compliance  
  review", "solution_steps": [ "Verify audit team identity", "Check approval  
  from compliance officer", "Grant temporary read-only access (24 hours)",  
  "Enable audit logging", "Notify security team" ], "estimated_time_minutes":  
  15, "requires_approval": true, "sop_reference": "SOP-789-Emergency-Access" }
```

Benefit: Easy to parse and integrate with downstream systems. Critical for production.

⚠️ Common Problems & Solutions

Problem 1: Hallucinations

The Issue

Model invents plausible-sounding but incorrect information.

User: "How do I fix null values in customer_transactions?"

LLM Response: "Run UPDATE customer_transactions SET amount = 0 WHERE amount IS NULL"

Problem: This could corrupt financial data! Model hallucinated a solution without checking company procedures.

Solutions

- **Constraint the domain:** "Only suggest solutions from these approved procedures: [list]"
- **Add safety check:** "If you're unsure, say 'I don't know' rather than guessing"
- **Require sources:** "Cite the documentation you're referencing"
- **Use RAG:** Move to Stage 4 for grounded responses (next guide)

Better Prompt

```
You are a support agent. IMPORTANT RULES: 1. Only suggest solutions from approved SOPs 2. If a solution is not in your knowledge, say "This requires L2 escalation" 3. NEVER suggest modifying production data without approval 4. Always include SOP reference number Available SOPs: - SOP-101: Schema sync procedures - SOP-202: Timeout configuration - SOP-303: Access requests Ticket: "How do I fix null values in customer_transactions?"
```

Result: Model will now admit when it doesn't know or escalate appropriately.

Problem 2: Inconsistent Outputs

The Issue

Same prompt gives different answers each time due to model randomness.

Run 1: "This is a critical issue. Fix immediately."
Run 2: "This is a minor issue. Fix when convenient."
Same Input!

Solutions

- **Lower temperature:** Set `temperature=0.0` for deterministic outputs
- **Explicit constraints:** "Rate severity as: critical/high/medium/low ONLY"
- **Use seed:** Some models support setting a random seed
- **Consider fine-tuning:** Move to Stage 5 for consistency (later guide)

```
# Configure for consistency
model = genai.GenerativeModel('gemini-1.5-pro',
    generation_config={
        "temperature": 0.0, # More deterministic
        "top_p": 0.1, # Less randomness
        "top_k": 1 # Most likely token
    })
```

Problem 3: High Token Costs

The Issue

Long prompts with many examples = high API costs at scale.

Scenario: 500 tickets/month, 2000 tokens per prompt
Cost: $500 \times 2000 = 1M$ tokens/month = \$3-5K/month
Problem: Not sustainable at scale

Solutions

- **Use smaller model:** Gemini Flash instead of Pro (cheaper, faster)
- **Reduce examples:** Find minimum examples needed (3 instead of 10)
- **Batch processing:** Process multiple tickets in one call
- **Cache prompts:** Use prompt caching for repeated content
- **Consider fine-tuning:** Lower per-query cost at high volume

Cost Optimization

```
# Use cheaper Flash model for simple classification
model_flash = genai.GenerativeModel('gemini-1.5-flash') # Batch multiple tickets
prompt = """Classify these 5 tickets: 1. {ticket1} 2. {ticket2} 3.
{ticket3} 4. {ticket4} 5. {ticket5} Return JSON array of
classifications.""" # Result: 5x throughput, lower cost per ticket
```



Best Practices for Production

1. Start Simple

Begin with zero-shot, add examples only if needed. Don't over-engineer.

2. Test Extensively

Run prompt on 50+ real tickets. Check for hallucinations, edge cases, failures.

3. Version Your Prompts

Track prompt changes like code. Use git. Document what works and what doesn't.

4. Monitor in Production

Log inputs, outputs, user feedback. Track hallucination rate, response quality.

5. Set Constraints

Explicitly tell model what NOT to do. Define allowed actions. Require escalation for edge cases.

6. Use Structured Outputs

Always request JSON/XML for downstream processing. Never parse free text in production.

Production Prompt Template

```
def create_production_prompt(ticket: str, user_context: Dict,
approved_sops: List[str]) -> str: """
Production-grade prompt with safety constraints.
"""
prompt = f"""
You are an L1 IT Support Agent for banking data platform. CRITICAL SAFETY RULES:
1. Only suggest solutions from approved SOPs: {', '.join(approved_sops)}
2. NEVER suggest modifying production data without explicit approval
3. If uncertain, respond with: "Requires L2 escalation"
4. Always include SOP reference in your response
5. Rate severity objectively: critical/high/medium/low
USER CONTEXT:
- Department: {user_context['department']}
- Access Level: {user_context['access_level']}
- Previous Tickets: {user_context['ticket_history_summary']}
RESPONSE FORMAT (JSON ONLY):
{{ "category": "string", "severity": "critical|high|medium|low", "root_cause": "string", "solution_steps": ["step1", "step2"], "estimated_minutes": integer, "sop_reference": "string", "escalate": "string" }}"
```

```
boolean, "escalation_reason": "string or null" }} TICKET: {ticket} Respond  
with ONLY the JSON object, no additional text.""" return prompt
```



When to Use GenAI Prompting

✓ Use GenAI Prompting When:

- **Rapid prototyping** — Need to test idea in days not weeks
- **Natural language understanding** — Users express issues in many ways
- **Low to medium volume** — <10K requests/day
- **Flexible requirements** — Task evolves frequently
- **No training data** — Can't collect 1000s of labeled examples
- **Non-critical applications** — Occasional errors are acceptable

✗ Don't Use GenAI Prompting When:

- **Zero tolerance for errors** — Financial transactions, medical decisions
- **High volume** — 100K+ requests/day (costs become prohibitive)
- **Latency critical** — Need <500ms response time
- **Need deterministic behavior** — Same input must always give same output
- **Highly specialized domain** — Model lacks domain knowledge (use fine-tuning)
- **Need source citations** — Must prove where information came from (use RAG)

Evolution Trigger: Why We Need Stage 4 (RAG)

The Critical Problem with Pure Prompting:

Scenario: User asks "What's the procedure for handling schema errors?"

GenAI Response: "Run the schema sync job, restart the pipeline, and verify the data..."

Problems:

1. ✗ No source citation — can't verify accuracy

2. ❌ Might be outdated — procedures changed last month
3. ❌ Could be hallucinated — model might have invented steps
4. ❌ No compliance trail — banking requires documented sources

What We Need: A way to ground responses in actual, current documentation with verifiable sources. → This is why Stage 4 (RAG) was created!

Next Steps

Continue Learning

- [Stage 4: RAG Implementation](#) (Next)
- [Back to Master Guide](#)
- [Quick Start Hub](#)

Hands-On Practice

- Get Gemini API key
- Try the code examples
- Experiment with prompts
- Build ticket classifier

Resources

- [Gemini API Docs](#)
- [Prompting Guide](#)
- [Code Examples](#)