1. **Zero-shot prompting**
2. **Few-shot prompting**
3. **Chain of thought prompting**
4. **Tree of Thoughts prompting**
5. **React-based prompting (Reasoning plus Acting)**
6. **JSON-based prompting**

**Zero short prompting**

Prompt: """ Classify the sentiment of this review as positive, negative, or neutral: "The API documentation was thorough but the examples were outdated." """ Expected Output: "Neutral" Now answer the following - "A boy ate an apple and felt fulfilling"

“Positive”

**Few-shot prompting**

Prompt: """ Extract key technical specifications from product descriptions: Example 1: Input: "Our cloud service offers 99.9% uptime with 24/7 support" Output: {"uptime": "99.9%", "support": "24/7"} Example 2: Input: "Database scales to 10TB with automatic backups every hour" Output: {"max\_storage": "10TB", "backup\_frequency": "hourly"} Now extract from: "API handles 10,000 requests/second with sub-50ms latency" """ Expected Output: {"throughput": "10,000 req/s", "latency": "<50ms"} Now answer this - Database scales to 15TB with automatic backups every 2 hours

**ChatGPT said:**

{"max\_storage": "15TB", "backup\_frequency": "2 hours"}

**You said:**

**Chain of thought prompting**

Prompt: """ A distributed system has 3 microservices. Service A calls Service B which calls Service C. Each service has 99.5% availability. What's the end-to-end availability? Show your reasoning step-by-step. """ Expected Output: """ Step 1: Each service must be available for the request to succeed Step 2: Availability = P(A) × P(B) × P(C) Step 3: Calculation = 0.995 × 0.995 × 0.995 Step 4: Result = 0.9851 or 98.51% """ Now answer this - A distributed system has 4 microservices. Service A calls Service B which calls Service C, D. Each service has 99.2% availability. What's the end-to-end availability?

**ChatGPT said:**

"""  
Step 1: Each service in the chain must be available for the request to succeed  
Step 2: Availability = P(A) × P(B) × P(C) × P(D)  
Step 3: Calculation = 0.992 × 0.992 × 0.992 × 0.992  
Step 4: Result ≈ 0.9684 or 96.84%  
"""

**You said:**

We need to optimize database query performance. Generate 3 different approaches, evaluate each, then recommend the best: Approach 1: I'm thinking of distributed database Pros/Cons: All I know is sharding, feel free to elaborate and make this more comprehensive. Approach 2: [Generate] Pros/Cons: [Evaluate] Approach 3: [Generate] Pros/Cons: [Evaluate] Final Recommendation: [Select best with reasoning]

**ChatGPT said:**

Thought for a second

Below are three viable approaches to improve database query performance, followed by an evaluation and a recommendation tailored to typical product constraints of cost, complexity, and time-to-impact.

**Approach 1: Distributed database with sharding and replication**

**What it is.** Horizontally partition data across multiple nodes (shards) and replicate for HA. Common patterns include range, hash, and geo sharding; secondary indexes may be local or global. Reads can be routed to replicas; writes go to shard primaries.

**Pros**

* Scales write and read throughput nearly linearly with additional shards.
* Isolates “hot” key ranges when using range sharding with careful key design.
* Enables geo-locality and lower latency for regional users.
* Failure domains are smaller per shard; maintenance can be rolling.

**Cons**

* Significant operational and logical complexity: cross-shard transactions, fan-out queries, rebalancing, hotspot avoidance, and global secondary index consistency.
* Application changes: shard routing keys, cross-shard joins become expensive or require data duplication.
* Higher cost and operational overhead for monitoring, backups, schema evolution, and incident response.
* Latency variance from scatter/gather plans for multi-shard reads.

**Best for**

* Very large datasets or sustained write QPS beyond a single node’s limits.
* Clear, stable shard keys with limited cross-entity joins.

**Approach 2: Single-cluster optimization: schema, queries, and indexing**

**What it is.** Keep the current deployment topology, but aggressively optimize within it.

**Tactics**

* **Indexing:** add/selective composite indexes, covering indexes, partial/filtered indexes; remove unused/overlapping indexes.
* **Query tuning:** rewrite N+1 patterns to set-based queries; push predicates; avoid wildcard leading patterns; proper pagination (keyset/seek over OFFSET); use prepared statements; batch writes.
* **Execution plans:** analyze with EXPLAIN/ANALYZE, fix misestimates via stats refresh, hints (where supported), or rewriting predicates to sargable forms.
* **Schema changes:** denormalize selectively for heavy read paths; precompute columns; choose appropriate datatypes; partition large tables (range/hash/list); archive cold data.
* **Physical layout:** tune fillfactor/clustered index order; reduce bloat; compress where supported; align I/O with SSDs; ensure enough RAM for working set.
* **Runtime:** right-size connection pooling; increase effective cache/buffer sizes; enable parallel query where safe.

**Pros**

* Fastest time-to-impact with minimal architectural risk.
* Low cost relative to alternatives.
* Improves both average and tail latency by eliminating plan/pathological cases.
* No major changes to application semantics.

**Cons**

* Diminishing returns once low-hanging fruit is addressed.
* Ceiling remains the limits of a single primary node.
* Requires disciplined performance hygiene to sustain gains.

**Best for**

* Most systems that have not yet been rigorously tuned.
* Workloads where 80/20 hotspots exist.

**Approach 3: Caching, read replicas, and precomputation (CQRS/materialized views)**

**What it is.** Offload read pressure and precompute expensive results.

**Tactics**

* **Caching layers:** Redis/Memory cache for hot keys, request/result cache with TTL, cache-aside or write-through; idempotent invalidations on writes.
* **Read replicas:** async replicas for read-heavy traffic; read/write split at the app/router layer; pin strongly consistent reads to primary where needed.
* **Precomputation:** materialized views, summary tables, or streaming pipelines to maintain aggregates; scheduled refresh or incremental updates.
* **CDN/edge caching:** for API responses that are public or user-agnostic.

**Pros**

* Large reductions in DB load and p99 latency for read-heavy endpoints.
* Incremental adoption per endpoint; limited schema churn.
* Enables near-real-time analytics without stressing OLTP tables.

**Cons**

* Cache invalidation complexity; risk of stale reads.
* Replica lag may violate strict consistency for some queries.
* Operational overhead for replication, failover, and view refresh.
* Does not increase write capacity on the primary.

**Best for**

* Read-heavy systems, expensive aggregations, or endpoints with stable responses.

**Comparative evaluation**

| **Criterion** | **Approach 1: Shard** | **Approach 2: Optimize** | **Approach 3: Cache/Replicas** |
| --- | --- | --- | --- |
| Time-to-impact | Low | High | Medium–High |
| Complexity | High | Low–Medium | Medium |
| CapEx/OpEx | High | Low | Medium |
| Read latency gains | Medium | High | High |
| Write scalability | High | Low–Medium | Low |
| Risk | High (data model & ops) | Low | Medium (staleness/lag) |

**Final recommendation**

Adopt a staged plan that maximizes near-term gains and preserves optionality.

1. **Start with Approach 2 (optimize single cluster).** Profile top 10 slowest/highest-QPS queries, add or fix composite/covering indexes, remove unused indexes, convert OFFSET to keyset pagination, batch writes, and partition any table with time-series growth. Expect material improvements within days.
2. **Layer in Approach 3 for heavy reads and aggregations.** Introduce Redis cache-aside for the hottest endpoints with strict TTLs and explicit invalidation on writes. Add 1–2 read replicas and route non-critical reads to them. For expensive analytics, create materialized views or summary tables refreshed incrementally.
3. **Only proceed to Approach 1 (sharding) when metrics show you are saturating a well-tuned primary or require regional locality.** Before sharding, validate shard keys against access patterns, inventory cross-entity joins, and build migration/rebalancing playbooks.

**Success metrics to track**

* p50/p95/p99 latency per endpoint, cache hit ratio, primary CPU/IO utilization, buffer cache hit rate, replica lag, top queries by total time, and error rates during deploys.

This path delivers rapid performance wins with minimal risk, while keeping a clear escalation route to horizontal scale if and when the workload necessitates it.

**You said:**

1. **JSON-based prompting**

Debug this Python error step-by-step: Error: "TypeError: unsupported operand type(s) for +: 'int' and 'str'" Code: result = age + "5" Thought 1: [Analyze error type] Action 1: [Identify problematic line] Observation 1: [What you found] Thought 2: [Determine root cause] Action 2: [Propose fix] Observation 2: [Expected outcome] Final Solution: [Corrected code]

**ChatGPT said:**

Thought for 11s

**Thought 1: [Analyze error type]**  
A TypeError occurs because the + operator is being applied to incompatible types.

**Action 1: [Identify problematic line]**  
Inspect the provided line:

result = age + "5"

**Observation 1: [What you found]**  
age is an int, while "5" is a str. Python does not support adding an integer to a string.

**Thought 2: [Determine root cause]**  
The operation mixes numeric addition with a string literal, causing a type mismatch.

**Action 2: [Propose fix]**  
Decide the intended behavior:

* If numeric addition is intended, convert "5" to an integer.
* If string concatenation is intended, convert age to a string.

**Observation 2: [Expected outcome]**  
After consistent typing, the expression will evaluate without error. For example, if age = 30, numeric addition yields 35; string concatenation yields "305".

**Final Solution: [Corrected code]**  
Numeric addition (likely intent):

result = age + 5

# or, explicitly:

# result = age + int("5")

Alternative, if concatenating text is desired:

result = str(age) + "5"

**You said:**

Extract API documentation into JSON: GET /users/{id} Description: Retrieves user profile Auth: Bearer token required Rate limit: 100/hour Response codes: 200 (success), 404 (not found), 401 (unauthorized) Return as JSON with fields: endpoint, method, description, auth\_required, rate\_limit, response\_codes

**ChatGPT said:**

{

"endpoint": "/users/{id}",

"method": "GET",

"description": "Retrieves user profile",

"auth\_required": true,

"rate\_limit": "100/hour",

"response\_codes": {

"200": "success",

"404": "not found",

"401": "unauthorized"

}

}

**ession Details and Highlights**

This session focused on building transformer-based applications with Gradio and covered several key topics:

Prompting Techniques

* Zero-shot prompting: Giving instructions without examples
* Few-shot prompting: Providing examples before asking for output
* Chain of Thought: Walking through reasoning step-by-step
* Tree of Thoughts: Exploring multiple solution approaches with pros and cons
* React-based prompting: Debugging through thought-action-observation cycles
* JSON-based prompting: Structuring outputs for API integration

Hugging Face Ecosystem

* Models: Repository of open-source and paid AI models
* Datasets: Collections of training data for various use cases
* Spaces: Deployed demo versions of models with interfaces
* Authentication with Hugging Face tokens for accessing models

Transformer Architecture

* Explained how transformers use attention mechanisms to understand relationships between words
* Demonstrated how models assign different levels of attention to words in sentences
* Covered encoder models (classification), decoder models (generation), and encoder-decoder models (summarization)

Tokenization and Embeddings

* Showed how text is broken down into tokens for model processing
* Explained how embeddings position words in multi-dimensional space
* Discussed token usage and its impact on model costs

Model Deployment Options

* Hosted inference: Using cloud providers like OpenAI, Hugging Face
* Local deployment: Running open-source models on your own hardware
* Quantized models: Compressed versions for local use with less computing power

Practical Exercise

* Breakout room assignment: Build a text summarization app using Gradio
* Requirements included implementing a text summarization model, creating a Gradio interface, adding export functionality, and customizing the theme
* Several participants demonstrated their solutions, including additional features like audio output and multi-model selection

The session combined theoretical concepts with hands-on coding, emphasizing that Gradio is ideal for prototyping but not recommended for production applications due to its size and performance characteristics.