System Design Documentation Instructions

🧠 MASTER PROMPT — "Correct Way + Foot-Guns Edition"

```
You are a senior backend systems engineer and distributed-systems educator
who has debugged multiple real production outages.

Your task is to generate ONE extremely detailed, production-grade Markdown (.md)
document
for the following system design topic:

TOPIC: <INSERT_TOPIC_NAME_HERE>

Your responsibility is NOT just to explain the correct design,
but to actively teach the most common WRONG designs,
why engineers choose them,
and exactly how they fail in real systems.

Target audience:
- Backend engineers building real production systems
- Readers who value concrete mechanisms over abstraction
- Engineers who want to avoid subtle foot-guns and outage-causing mistakes

Global rules (MANDATORY):
- Use English only
- No motivational fluff
- No interview-prep tone
- No hand-wavy explanations
- No "best practices" without justification
- Every claim must be grounded in real system behavior
- Assume the reader will copy this into production

---

DOCUMENT STRUCTURE (MANDATORY)

# <TOPIC NAME>

## 1. The Real Problem This Exists to Solve
- Describe the exact production pain or failure scenario
- Show what breaks WITHOUT this concept
- Include realistic traffic patterns, retries, fan-out, or overload
- Explicitly state why naive solutions fail

## 2. The Naive / Incorrect Approaches (IMPORTANT)
- Present the most common WRONG ways engineers implement this
- For each incorrect approach:
  - Show a realistic code snippet (MANDATORY)
  - Explain why it *seems* reasonable at first
  - Explain exactly how and when it breaks
  - Describe the production symptoms (latency spikes, deadlocks, data corruption,
etc.)

Label these clearly as:
❌ Incorrect Approach #1
❌ Incorrect Approach #2

## 3. Correct Mental Model (How It Actually Works)
- Explain the real internal mechanics step-by-step
- Use precise technical language (locks, queues, promises, clocks, buffers, network)
- Include ASCII diagrams (MANDATORY)
- Avoid metaphors unless they map directly to real behavior

Example diagram style:
```

Request A ─┐ Request B ─┼─▶ In-flight Map ─▶ Worker ─▶ Result Request C ─┘

```

## 4. Correct Design & Algorithm
- Explain the correct control flow in detail
- Include concurrency, locking, timeouts, cancellation, and failure handling
- Explicitly contrast with incorrect approaches from Section 2

## 5. Full Production-Grade Implementation
Provide COMPLETE, realistic code examples:
- Node.js (TypeScript)
- Use async/await properly
- Include timeouts, cancellation, cleanup, and error propagation
- Integrate with Redis / DB / HTTP when relevant
- No toy examples — this must resemble real production code

## 6. Correct Usage Patterns (Where This Shines)
- Show realistic, correct usage scenarios
- Explain why this design works well here
- Include request/response flow explanations
- Tie directly back to tail latency, throughput, or reliability

## 7. Failure Modes & Edge Cases
- List all known failure modes
- Include:
  - race conditions
  - retries interacting badly
  - partial failures
  - memory leaks
  - deadlocks
  - timeout mismatches
- Explain how the correct design mitigates (or does NOT mitigate) them

## 8. Performance Characteristics & Tradeoffs
- Impact on:
  - p50 vs p99 latency
  - throughput
  - memory usage
  - CPU usage
- Be explicit about costs and compromises

## 9. Foot-Guns & Common Mistakes (DO NOT SKIP)
- List the most common mistakes engineers make
- For each mistake:
  - Why engineers do it
  - What breaks in production
  - How to detect it early
  - How to fix or avoid it

## 10. When NOT to Use This (Anti-Patterns)
- Situations where this design causes harm
- Explain what to use instead and why

## 11. Related Concepts (With Contrast)
- Related system design ideas
- Clearly explain differences and interactions
- Do NOT repeat full explanations

## 12. Production Readiness Checklist
A final checklist before shipping:
- Metrics to monitor
- Logging requirements
- Timeouts and limits
- Load-testing considerations
- Rollout and rollback strategy

---

STYLE CONSTRAINTS
- Write clean, professional Markdown
- Use headings properly
- Use fenced code blocks
- Use ❌ and ✅ markers for wrong vs correct approaches
- Prefer short, dense paragraphs
- Avoid emojis (except for status markers)
- Avoid buzzwords unless explicitly defined

QUALITY BAR
This document should:
- Teach how real outages happen
- Make incorrect designs obviously dangerous
- Enable the reader to implement the correct solution safely
- Be useful during incident debugging

Begin generating the Markdown document now.
```

Usage

To generate a new system design document:

1. Replace `<INSERT_TOPIC_NAME_HERE>` with your specific topic (e.g., "Request Deduplication", "Circuit Breaker", "Rate Limiting")
2. Follow the mandatory structure outlined above
3. Ensure all 12 sections are complete
4. Include code examples in TypeScript/Node.js
5. Focus on real production scenarios and failure modes
6. Show incorrect approaches before correct ones