# How to Use the Complete Code Design Prompt Template

This guide explains **exactly how to use the template effectively**, step by step, without adding friction to your workflow.

---

## 1. Start With One Thing Only

Do **not** try to fill the entire template at once.

Begin by filling **only**: - **My Requirement** - **SECTION 1: Problem Understanding**

This forces clarity before architecture and prevents premature design decisions.

---

## 2. Use the Template as a Prompt Contract

When using this template with ChatGPT or any LLM:

1. Paste the **entire template**
2. Fill **only the "My Requirement" section** initially
3. Explicitly instruct:

   "Use the provided template structure strictly. Do not skip sections."

This prevents free-form answers and enforces structured system thinking.

---

## 3. Build the Design Incrementally (Critical)

Use the template across **multiple iterations**, not in one go.

### Iteration 1: System Boundaries

- SECTION 1: Problem Understanding
- SECTION 2: Data Flow Diagram
- SECTION 3: Component Identification

Focus: *What exists and how data moves*

---

### Iteration 2: Internal Design

- SECTION 4: Data Structures
- SECTION 5: Detailed Component Design
- SECTION 6: Complete Data Flow (Step-by-Step)

Focus: *How components work internally*

---

### Iteration 3: Architecture & Execution

- SECTION 7: Component Interaction Diagram
- SECTION 8: Data Origin Mapping
- SECTION 9: File Structure
- SECTION 10: Implementation Order
- SECTION 12: Threading / Async Analysis

Focus: *Execution, ownership, and concurrency*

---

### Iteration 4: Quality & Scale

- SECTION 11: Complete Code Skeleton
- SECTION 13: Error Handling Strategy
- SECTION 14: Testing Approach
- SECTION 15: Best Practices & Pitfalls
- SECTION 16: Extension Guide

Focus: *Reliability, testing, and future growth*

---

## 4. Treat Sections as Design Gates

Each section is a **gate**.

If a section feels unclear, **stop and fix it before moving forward**.

Examples: - If the Data Flow Diagram is confusing, your trigger or boundaries are unclear - If components overlap, responsibilities are poorly defined

This strictness is intentional and prevents architectural debt.

---

## 5. Best-Fit Use Cases

This template is ideal for: - Backend system design - Event-driven architectures - Notification engines - PBX and telephony orchestration - Workflow engines - LLM-based toolchains - Async and distributed systems

Not recommended for: - Small CRUD-only features - Pure UI components

---

## 6. How to Use This Template With ChatGPT

Use clear, scoped instructions.

### Example 1

> "I want to use the Complete Code Design Prompt Template. Here is my filled ‘My Requirement’ section only. Walk through Sections 1–3 first."

### Example 2

> "Fill Sections 4–6 only for the following system. Do not repeat earlier sections."

This avoids duplication and keeps responses focused and high quality.

---

## 7. Recommended Working Style

- Treat this document as a **living design artifact**
- Revise earlier sections as understanding improves
- Do not lock code until Sections 1–3 are solid

This mirrors how senior engineers design production systems.

---

## 8. Suggested Next Step

Choose **one real system** you are already working on.

Fill **only the “My Requirement” section** and then proceed incrementally.

Strong candidates: - Notification Engine - Asterisk PBX and LiveKit routing - Timesheet activity crunching service - LLM orchestration or tool execution system

---

## 9. Outcome If Used Correctly

By following this guide, you will get: - Clear system boundaries - Predictable data flow - Minimal refactoring later - Production-ready architecture - Easier onboarding for new engineers

---

Use this guide alongside the main template as your **operating manual for system design**.

---

## 10. How to Get the Idea? (Discovery Process)

Use this section when you are **confused, starting from zero, or dealing with unknown data**. These methods help you *discover* what to build before you design it.

| HOW | TO | DISCOVER | WHAT | YOU | NEED |
|---|---|---|---|---|---|

### METHOD 1: READ DOCUMENTATION

——————————————————————— - For binlog or database events: Read **PostgreSQL logical replication / WAL** documentation - For APIs: Read the official **API documentation** - For libraries or SDKs: Read the **library docs and examples**

**This tells you:** - Input format - Available methods - Expected output - Limitations and guarantees

---

### METHOD 2: PRINT / LOG EVERYTHING

——————————————————————— If you do not know what the data looks like, **log it immediately**.

```
print(msg.payload)              # Raw input
print(type(msg.payload))        # Data type
print(json.loads(msg.payload))  # Parsed structure
```

This is the **fastest way** to understand real-world data formats.

---

### METHOD 3: WORK BACKWARDS

——————————————————————— Start from the **final result** and trace backwards step by step.

Example:

```
"Send notification when ticket is created"
        ↑
"Need ticket data (id, title, assigned_to)"
        ↑
"Detect INSERT on tickets table"
        ↑
"Listen to PostgreSQL WAL"
        ↑
"Use LogicalReplicationConnection"
```

This prevents overengineering and keeps design outcome-driven.

---

### METHOD 4: BREAK INTO SMALL PIECES

——————————————————————— Convert a big problem into **solvable atomic steps**.

Example:

1. Connect to PostgreSQL WAL
2. Receive a single message
3. Parse the message
4. Identify table and event type
5. Route to correct handler
6. Handler performs action (notification, cache invalidate, sync)

Solve **one piece at a time**. Never design everything at once.

---

### METHOD 5: LOOK AT EXISTING CODE

———————————————————————— - Search GitHub for similar implementations - Read official library examples - Inspect existing code in your own codebase

Patterns already solved by others save time and reduce mistakes.

---

## 11. Universal Planning Template (Questions for ANY Task)

Use this checklist before writing **any code**, regardless of language or framework.

| UNIVERSAL | PLANNING | TEMPLATE |
|---|---|---|

### 1. INPUT

☐ Where does the data come from? ☐ What triggers this code to run? ☐ What is the raw input format? (string, bytes, JSON, object) ☐ What fields or properties are available?

---

### 2. PROCESSING

☐ What needs to happen to the data? ☐ Do formats need parsing or conversion? ☐ Are external resources required? (DB, API, cache) ☐ What business rules apply?

---

### 3. OUTPUT

☐ Who or what receives the output? ☐ What format do they expect? ☐ What data must be included?

---

### 4. ERRORS

☐ What can fail? ☐ How should each failure be handled? ☐ Retry, log, alert, or fail fast?

---

### 5. STRUCTURE

☐ Should this be a function or a class? ☐ What parameters are required? ☐ What should be returned?

---

**Use this section as your mental debugger before design or coding begins.**

---

## 12. The 7 Methods to Understand Any Code (Learning Companion)

This section is **not part of the core design template**. It is a **personal learning and debugging companion** you should keep alongside the template for day-to-day engineering work.

```
| 7 WAYS TO UNDERSTAND ANY CODE | | (From Most to Least Effective) |
```

### Method 1: TRACE THE ENTRY POINT (Most Effective)

Every program has an **entry point**. Find it first and follow the flow.

- Web apps (FastAPI / Flask): `main.py` → app startup → routes
- CLI tools: `if __name__ == "__main__"`
- Background services: `start()` function or main loop
- Libraries: `__init__.py` exports

**Example (Binlog System):** 1. `main.py` calls `start_binlog_monitor()` 2. Function creates `PostgresWALMonitor` 3. Calls `.start()` 4. Follow each method call step by step

This gives you the **complete execution flow**.

---

### Method 2: PRINT / LOG EVERYTHING

If you do not understand a variable, **print it**.

```python
def some_confusing_function(data):
    print("=== ENTERING some_confusing_function ===")
    print(f"Input: {data}")
    print(f"Type: {type(data)}")

    result = do_something(data)
    print(f"After do_something: {result}")

    return result
```

Fastest way to understand: - Real data shape - Data types - Transformations - Code paths

---

### Method 3: USE DEBUGGER (Step Through Code)

In VS Code: 1. Click left of line number to set breakpoint 2. Press **F5** to start debugger 3. Use: - F10: Step over - F11: Step into - F5: Continue 4. Hover over variables to inspect values

Best for: - Complex logic - Execution order - Subtle bugs

---

### Method 4: READ BACKWARDS (Output → Input)

Sometimes it is easier to start from the **end result**.

Example: 1. Find `send_notification()` 2. Who calls it? `notify_users()` 3. Who calls that? `on_ticket_insert()` 4. Who triggers it? `Router.dispatch()` 5. Who calls router? `PostgresWALMonitor._emit()`

Now you have the **entire call chain**.

---

### Method 5: DRAW DIAGRAMS

Visualize what you read.

```
main.py → binlog.py → monitor → router → handlers
```

Tools: - Paper & pen - draw.io - Excalidraw - Mermaid

Diagrams reveal architecture faster than text.

---

### Method 6: RENAME THINGS MENTALLY

Bad code often has bad names. Rename them **in your head**.

Confusing code:

```
def proc(d, f):
    x = f()
    for i in d.get("c", []):
        k = i.get("k")
```

Mental rename:

```
def process_wal_message(payload, db_factory):
    db_session = db_factory()
```

```
    for change in payload.get("changes", []):
        event_kind = change.get("kind")
```

Add comments with **your understanding**.

---

### Method 7: ISOLATE AND TEST SMALL PIECES

Copy confusing logic and test it alone.

```
def combine(names, values):
    return dict(zip(names, values))

sample = {"names": ["id", "title"], "values": [1, "test"]}
print(combine(sample["names"], sample["values"]))
```

Isolation leads to clarity.

---

## 13. The 30-Minute Code Understanding Method

### Minute 0–5: Overview

- Scan folder structure
- Read README
- Identify entry file

### Minute 5–10: Entry Point

- Where does execution start?
- What initializes first?

### Minute 10–20: Follow One Flow

- Pick one feature
- Trace start to end
- Add prints if needed

### Minute 20–25: Draw It

- Boxes for components
- Arrows for data flow

### Minute 25–30: Verify

- Run code
- Confirm assumptions

---

## 14. Quick Reference: What to Do When Stuck

| Problem | Solution |
|---|---|
| Don’t know where to start | Search for `main.py`, `app.py`, `__main__` |
| Don’t know variable content | `print(var)` and `print(type(var))` |
| Don’t know which path runs | Add prints in branches |
| Don’t understand function | Copy and test separately |
| Too many files | Follow ONE flow only |
| Confusing names | Rename mentally, add comments |
| Lost in flow | Draw diagram |
| Unexpected behavior | Use debugger |

---

## 15. The Golden Rule

```
| "If you can’t explain what a piece of code does, | | you haven’t understood it yet." | | | | Test:
Can you explain it to someone else? | | YES → You understand it | | NO → Print, debug, and trace
again                                                                                          |
```