
## Module 1: The AI-Augmented Environment, Lesson 3: The R-I-C-E Framework

## Hands-On: R-I-C-E in Practice

## Overview

This demo proves three claims from the R-I-C-E lecture in real time:

- A vague prompt scatters the model's attention and costs you hours of rework
- A R-I-C-E-structured prompt concentrates attention and yields deterministic output
- R-I-C-E only pays off at scale if you turn it into a one-keystroke IDE snippet

You will run the same PostgreSQL optimization task through three prompts: first without structure, then with full R-I-C-E, then you will have the AI generate a reusable R-I-C-E snippet you can save to your editor today.

---

## Before You Start

- Open your AI tool of choice: Claude.ai, Cursor, or GitHub Copilot Chat
- Open an empty scratch file in VS Code or your editor where you can paste SQL output
- Keep the R-I-C-E four-letter reminder visible on screen or a second monitor
- Do NOT start a new chat between prompts 1 and 2 — we want to see the contrast side by side

---

> **Remember**
> Every good AI coding prompt has exactly four blocks: Role, Instructions, Context, Expectations. The 30 seconds you spend writing them buys you deterministic, pipeline-ready output and eliminates the three hours of debugging that a vague prompt guarantees.

## During Hands-On — Keep Visible if Possible

```
R   — ROLE          — assign hyper-specific domain expertise
I   — INSTRUCTIONS  — numbered, imperative, one action per step
C   — CONTEXT       — versions, schema, conventions, ground truth
E   — EXPECTATIONS  — output format, safety rules, scope limits
```

---

## Prompt 1 — Feel the Pain (The Vague Prompt)

> **What to Learn**
> This is the prompt 90% of developers actually type. No role. No steps. No schema. No output constraint. Watch how the AI, left with zero boundaries, hallucinates a plausible environment and proposes sweeping refactors you never asked for. This is not the AI's fault — it is doing exactly what your vague input implied.

**Paste this into the AI chat:**

```
Optimize this database query for the user dashboard:

SELECT u.id, u.email, u.name,
       COUNT(DISTINCT o.id) AS order_count,
       SUM(o.total)         AS lifetime_value,
       MAX(o.created_at)    AS last_order_date
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE u.is_active = true
GROUP BY u.id, u.email, u.name
ORDER BY lifetime_value DESC
LIMIT 100;
```

> **What you should get**
> A sprawling, multi-paragraph response. The AI will typically suggest adding a denormalized lifetime_value column, introducing a Redis cache, rewriting the ORM layer, and — most dangerously — creating or dropping indices without knowing what already exists. You will see verbose prose explaining concepts like "covering indices" that you already understand. Almost nothing is directly pasteable into a migration file.

**You should ask:**
"Which of the AI's suggestions is safe to deploy to production right now?"

Answer: none of them. Every recommendation is conditional on assumptions the AI made up because we refused to supply them. That is the $1,000 mistake from the lecture, reproduced in front of you in under a minute.

## Prompt 2 — Apply R-I-C-E (The Precision Prompt)

> **Remember**
> Same chat window, same task, same slow query. The only variable changing is the prompt structure. Watch how each R-I-C-E block prunes a specific category of hallucination: Role prunes generic advice, Instructions prune scope creep, Context prunes fabricated environments, Expectations prune verbose prose.

**Continue in the same chat session — do NOT start a new one:**

```
ROLE:
Act as a Principal PostgreSQL Architect specializing in read-heavy
OLTP query optimization for production environments with >10M
rows per table.

INSTRUCTIONS:
1. Analyze the EXPLAIN ANALYZE output below and identify every
   sequential scan.
2. Identify the single highest-cost node in the query plan.
3. Write the migration SQL that resolves that one bottleneck.
4. Do NOT modify existing indices. Do NOT alter the ORM layer.
   Do NOT suggest schema denormalization or caching.

CONTEXT:
- PostgreSQL 15.4, read replicas enabled, snake_case naming
- Schema:
    users  (id BIGSERIAL PK, email TEXT UNIQUE, name TEXT,
            is_active BOOLEAN)
    orders (id BIGSERIAL PK, user_id BIGINT FK->users(id),
            total NUMERIC(10,2), created_at TIMESTAMPTZ)
- Existing indices (do NOT touch):
    idx_users_is_active    ON users(is_active)
    idx_orders_user_id     ON orders(user_id)
- EXPLAIN ANALYZE output:
  Limit  (cost=45234.12..45234.37 rows=100)
    -> Sort  (cost=45234.12..45467.89 rows=93508)
         Sort Key: (sum(o.total)) DESC
         -> HashAggregate  (cost=43102.34..44037.42 rows=93508)
              Group Key: u.id
              -> Hash Left Join  (cost=3421.00..38145.22 rows=991422)
                   -> Seq Scan on users u  (cost=0.00..2145.00)
                        Filter: is_active
                   -> Seq Scan on orders o  (cost=0.00..1823.00)

EXPECTATIONS:
```

```
- Output only the raw SQL migration script wrapped in BEGIN/COMMIT.
- Script must be idempotent (use CREATE INDEX IF NOT EXISTS).
- Add a one-line SQL comment above each statement stating its purpose.
- Suppress all prose, commentary, and explanation outside the SQL block.
```

> **What you should get**
> A single fenced SQL block wrapped in BEGIN/COMMIT. One or two CREATE INDEX IF NOT EXISTS statements targeting orders(user_id, created_at) or a covering index that eliminates the sequential scan. Every statement has a one-line comment above it. Zero prose outside the block. The migration is pasteable into Flyway or Alembic without a single edit.

**You should ask:**
“If I remove any one of the four R-I-C-E blocks, which one opens the biggest hallucination vector?”

Answer: Context. Without the schema, version, and existing indices pasted in, the AI cannot know what it must not touch. Role and Expectations shape style; Context grounds the output in your actual system. Drop it and you are back to Prompt 1 even if every other block is perfect.

*Tip: Notice how you wrote zero English explanation of the problem. You described structure, not intent. That is the attention-mechanism trick — structure out-weighs narration every time.*

## Prompt 3 — Operationalize (Save R-I-C-E as a Snippet)

> **What to Learn**
> R-I-C-E only becomes a skill when typing the scaffold costs you zero seconds. If you have to reconstruct four blocks from memory every time, you will skip it under deadline pressure. The fix is to make the scaffold a keystroke. We will have the AI write the snippet for us — a mini R-I-C-E prompt whose output is itself a R-I-C-E template.

**New chat OK for this one — it is a different task:**

```
ROLE:
Act as a senior developer-productivity engineer who ships IDE
tooling for engineering teams.

INSTRUCTIONS:
1. Generate a VS Code user snippet in JSON format.
2. Name the snippet trigger "rice" and give it a description field
```

```
   that reads "R-I-C-E prompt scaffold".
3. The snippet body must expand into four labeled blocks in order:
   ROLE, INSTRUCTIONS, CONTEXT, EXPECTATIONS.
4. Place a numbered tab stop inside each block ($1, $2, $3, $4) so
   I can tab through and fill them sequentially.
5. Inside each tab stop, include a short placeholder hint showing
   what kind of content belongs there.

CONTEXT:
- Target editor: VS Code 1.90+ global user snippets
  (Code > Settings > Configure User Snippets > New Global Snippets File).
- The snippet must work in every language ("scope" omitted or "*").
- JSON syntax must be valid and parseable with no trailing commas.

EXPECTATIONS:
- Output only the raw JSON object.
- No markdown code fences. No prose before or after the JSON.
- Field order: prefix, description, body.
- The "body" value must be an array of strings, one per line.
```

> **What you should get**
>
> A single JSON object roughly 15 lines long. Key "R-I-C-E Prompt" (or similar) wrapping prefix "rice", a description, and a body array whose lines expand into the four R-I-C-E headers with $1 / $2 / $3 / $4 tab stops. You paste it straight into your global.code-snippets file, hit save, and from that moment forward typing “rice” in any file tab-expands the full scaffold.

**Install it now:**

- In VS Code, open the Command Palette (Cmd+Shift+P / Ctrl+Shift+P)
- Run “Snippets: Configure Snippets”, then “New Global Snippets file” and name it rice.code-snippets
- Paste the JSON the AI produced, save the file, open a new scratch file, type rice, press Tab
- The scaffold expands. You are now one keystroke from R-I-C-E on every future prompt.

*Tip: If you use Cursor or JetBrains AI, ask the same prompt but change the CONTEXT block to target that editor’s snippet / live-template syntax. The R-I-C-E structure ports; only the target grammar changes.*

## Wrap-Up — The Pattern, The Assignment Handoff

> **Remember**
> Three prompts. One chat. You felt the pain of a vague prompt, you watched R-I-C-E collapse the hallucination surface to near zero, and you walked away with a one-keystroke scaffold that will follow you to every project from today onward. That is the entire arc of this module — a measurable, compounding shift from conversational AI use to engineered AI use.

**Key takeaways:**

- R-I-C-E is four blocks, not a suggestion: Role, Instructions, Context, Expectations
- Role prunes the search space; a specific title beats a generic one every time
- Instructions are algorithms — numbered, imperative, one action per step, explicit NOTs
- Context is where seniors leave value on the table — versions, schema, conventions, existing state
- Expectations dictate output shape — format, safety, scope — so results are pipeline-ready
- Operationalize R-I-C-E as an IDE snippet on day one or it will not survive your first deadline