Module 1: Foundations of AI-Augmented Engineering, Lesson 4: Context is King

Hands-On Capstone: From Zero to AI-Native — Setup, Compare, and Engineer Your First Prompt

---

Overview

This minute capstone proves four claims from Module 1 in real time:

- Your IDE can become an AI-native cockpit in minutes — not months.
- The four major models (ChatGPT, Grok, Gemini, Claude) give meaningfully different answers to the same prompt, and knowing which to reach for is a skill.
- The R-I-C-E framework — Role, Instructions, Context, Examples — turns a one-line throwaway prompt into production-grade output.
- Context is not a vibe. It is measurable. We will build a Python tool that scores any prompt on R, I, C, and E using Claude.

You will install VS Code and Cursor with AI extensions, compare four models side-by-side, watch a single prompt improve through four revisions, and ship a working R-I-C-E Analyzer — all in one session.

Before You Start

- Create free accounts (if you don't already have them): ChatGPT, Grok (on x.com), Google Gemini, Claude.ai.
- Have Node.js 20+ and Python 3.11+ installed. Verify with node --version and python3 --version.
- Have an Anthropic API key ready in an environment variable: export ANTHROPIC_API_KEY=sk-ant-...
- Download installers for VS Code (code.visualstudio.com) and Cursor (cursor.com) — have them unopened, ready to install live.
- Open all four chat UIs in separate browser tabs before we start Segment 2.

During Hands-On — Four Segments at a Glance

1. SEGMENT 1 — Set up your AI-augmented environment (VS Code + Cursor) — ~6 min

2. SEGMENT 2 — Compare ChatGPT, Grok, Gemini, and Claude on 3 prompts — ~6 min
3. SEGMENT 3 — R-I-C-E in action: four progressive prompt revisions — ~7 min
4. SEGMENT 4 — Build and run the R-I-C-E Analyzer in Python — ~6 min

> **Remember**
> Module 1 is not about shipping a full-stack app. It is about wiring up the workshop — the cockpit, the models, the prompting grammar, and the measuring stick. Everything from Module 2 onward assumes you can install, configure, compare, and score. Today's deliverable is the workshop itself.

Segment 1 — Set Up Your AI-Augmented Environment

> **What to Learn**
> There is no single "best" AI-native editor. VS Code is the industry-standard extension host; Cursor is a VS Code fork that bakes AI into the core loop. A professional AI-native engineer keeps both installed and picks per-task. We will install both, configure both, and run a two-line query in each.

Part A — VS Code + AI Extensions

Step 1. Install VS Code. Download from code.visualstudio.com, accept defaults, open it.

Step 2. Install three AI extensions (Extensions panel, Ctrl/Cmd+Shift+X):

- GitHub Copilot — inline autocomplete + chat; works with a GitHub account.
- Continue — open-source, multi-model chat panel; lets you swap between Claude, GPT, Gemini, local models without leaving the editor.
- Claude Code for VS Code — agentic coding assistant from Anthropic; opens a side panel that can read, write, and run commands.

Step 3. Configure Continue for multi-model (UI path). Click the Continue icon in the left activity bar (the vertical strip on the far left of VS Code where Explorer and Extensions live). The chat panel opens on the right. Then:

1. Near the top of the chat panel, click the assistant selector dropdown.
2. Click "+ Add Chat Model."
3. Choose a Provider (Anthropic), then a Model (Claude), paste your API key, click Connect.

4. Repeat once more for OpenAI / GPT-4o (or any second provider you have a key for).

Once two models are configured, the model selector at the top of the chat panel becomes a dropdown — that is your A/B switch for Segment 2.

> **Advanced Alternative — Edit config.yaml Directly**
>
> Continue replaced `config.json` with `config.yaml`. If you would rather show the raw config file on camera, open the Command Palette (`Ctrl/Cmd+Shift+P`), run `Continue: Open Config File`, and replace the contents of `~/.continue/config.yaml` with the YAML below. Continue auto-reloads on save.

```
name: Multi-Model Setup
version: 1.0.0
schema: v1
models:
  - name: Claude Opus
    provider: anthropic
    model: claude-opus-4-7
    apiKey: sk-ant-...        # paste your Anthropic key
    roles:
      - chat
      - edit
  - name: GPT-4o
    provider: openai
    model: gpt-4o
    apiKey: sk-...            # paste your OpenAI key
    roles:
      - chat
      - edit
```

Step 4. Smoke test. Open the Continue chat panel, type:

```
Explain what this workspace does in 2 sentences.
```

Switch the model dropdown from Claude to GPT and re-run. You should see two visibly different answers to the same question — that is the multi-model muscle you'll use in Segment 2.

Part B — Cursor

Step 1. Install Cursor. Download from cursor.com. Open it and sign in.

Step 2. Import VS Code settings. On first launch, Cursor offers to import your VS Code keybindings, themes, and extensions. Accept it — the muscle memory carries over.

Step 3. Configure models. Settings → Models. Enable Claude, GPT, and Gemini. Optionally paste your own API keys under "Use your own key" for unlimited usage at cost price.

Step 4. Learn the three shortcuts that matter:

- **`Cmd/Ctrl+K`** — inline edit. Select code, press Cmd+K, describe the change. Cursor rewrites in place.
- **`Cmd/Ctrl+L`** — chat panel. Ask a question; drop files into the chat to add them to context.
- **`Cmd/Ctrl+I`** — Composer. Multi-file edits and whole-feature scaffolds. This is the real power move.

Step 5. Smoke test. Create a new file hello.py with:

```
def greet(name):
    pass
```

Select the function, press Cmd/Ctrl+K, type: implement this to return a friendly greeting with the current time. Cursor writes it in place. Accept with Enter.

> **What you should get**
> VS Code with three AI extensions installed and a multi-model config for Continue. Cursor running, signed in, with Claude/GPT/Gemini enabled and a working hello.py you edited via Cmd+K. Two editors, three shortcuts, zero excuses.

Segment 2 — Four Models, Three Prompts

> **What to Learn**
> Different models are not interchangeable. They were trained on different data, tuned with different objectives, and wrap their outputs in different personalities. We will run three carefully chosen prompts through all four frontier chat UIs — ChatGPT, Grok, Gemini, Claude — and watch the differences land.
>
> Open one browser tab per model. Keep them side-by-side. Paste each prompt exactly as written into all four.

Prompt A — Code Reasoning

Paste this identical prompt into ChatGPT, Grok, Gemini, and Claude:

```
Is there anything wrong with this Python function?
If so, how would you fix it?

def find_duplicates(lst):
    duplicates = []
    for item in lst:
        if lst.count(item) > 1:
            duplicates.append(item)
    return duplicates
```

What you should notice

- All four models should catch the $O(n^2)$ complexity — .count() inside a loop — and the fact that duplicates appear multiple times in the result.
- Claude typically leads with a structured breakdown (issues, then fix, then why) and produces a Counter-based one-liner.
- ChatGPT usually returns a polished, textbook answer with type hints and a short docstring.
- Gemini tends to be more verbose and often suggests multiple alternatives side-by-side.
- Grok is the most conversational and fastest to quip — useful for casual exploration, less useful when you need a contract-ready function.
- The takeaway is not "which is best" — it is that the same input produces four different artifacts. Pick per task.

Prompt B — Creative / Style

Paste this into all four:

```
Write a 5-line limerick about a senior engineer
debugging a memory leak at 3 AM. Make it funny.
```

What you should notice

- Meter and rhyme discipline vary dramatically. Claude and ChatGPT usually hit the AABBA scheme cleanly; Gemini sometimes drifts off meter; Grok tends toward edgier, funnier punchlines that occasionally break the form.
- This prompt reveals each model's "personality layer" — the tone and voice baked in by RLHF. When you are generating docs, commit messages, or user-facing copy, personality matters.

- Rule of thumb from this prompt alone: reach for Grok when you want voice; reach for Claude when you want craft; reach for ChatGPT when you want safe polish; reach for Gemini when you need integration with the Google stack.

Prompt C — Structured Planning

Paste this into all four:

```
I have 500 GB of plain-text customer-support chat logs.
I want to identify the top 5 recurring issues customers
report. Give me a step-by-step plan using only open-source
tools. Include data-privacy considerations.
```

What you should notice

- Claude typically asks a clarifying question first (format? PII? on-prem or cloud?) or sandwiches its plan with caveats about sensitive data — the safety-first training shows.
- ChatGPT produces the most textbook-linear plan: ingest → clean → embed → cluster → label — often with specific tool names (DuckDB, spaCy, BERTopic, HDBSCAN).
- Gemini leans toward Google-adjacent open source (TensorFlow, BigQuery's free tier) even when you asked for open source only; note the bias.
- Grok is the most opinionated and the shortest — it will pick a stack and defend it rather than enumerate options.
- Across all three prompts you have just performed the most important Module 1 reflex: trusting no single model for any single task.

Segment 3 — The R-I-C-E Framework, Live

What to Learn

R-I-C-E is the four-part grammar of every good prompt:

- **R = Role** — who the AI should be (senior engineer, security reviewer, tutor)
- **I = Instructions** — explicit tasks, constraints, and success criteria
- **C = Context** — the surrounding world (codebase, business, users, stack)

- **E = Examples** — concrete input/output pairs or reference material

We will take one realistic task — “write a user-registration validator” — and revise the prompt four times. Paste each version into Claude. Watch the output quality climb with every addition.

Prompt v1 — Naive (no R, no I, no C, no E)

```
Write a validation function for user registration.
```

What you should get

A generic 10-line function that checks maybe email format with a trivial regex and a minimum password length. No username rules. No structured error reporting. No security considerations. Works for a homework problem; unusable in production.

Prompt v2 — Add ROLE

```
You are a senior backend engineer who specializes in secure,
production-grade authentication systems.

Write a validation function for user registration.
```

What you should get

Noticeably more considered output. The function now typically covers email, password strength, and sometimes username. Comments mention security. Still no specific rules, no error-shape contract, no domain fit — the role alone pushes quality up ~30% with five added words.

Prompt v3 — Add INSTRUCTIONS

```
You are a senior backend engineer who specializes in secure,
production-grade authentication systems.

Write a JavaSCript validation function for user registration
that:

  - validates email format (RFC 5322 compliant regex)
  - requires password: 12+ chars, at least 1 uppercase, 1
```

```
    lowercase, 1 digit, and 1 special character
  - validates username: 3-20 chars, alphanumeric + underscore only
  - returns an object { "valid": bool, "errors": list[str] }
  - uses type hints throughout
  - never raises exceptions for validation failures
```

What you should get

Output now matches your spec field-for-field. The function signature is typed, the return shape is exactly what you asked for, each rule is enforced explicitly. You could drop this into a code review with minor adjustments. But the error messages are still generic ("Invalid email") and the function doesn't know it's running inside a FastAPI middleware for a B2B SaaS — that's the context gap we'll close next.

Prompt v4 — Add CONTEXT and EXAMPLES

```
You are a senior backend engineer who specializes in secure,
production-grade authentication systems.

CONTEXT:
This function runs inside an HTTP middleware for our Express.js
backend serving a B2B SaaS platform. Rejected fields must give
users actionable, plain-English feedback — never error codes,
never stack traces. Logs must not contain passwords.

INSTRUCTIONS:
Write a Node.js 20+ validation function for user registration
that:
- validates email format (RFC 5322 compliant regex)
- requires password: 12+ chars, at least 1 uppercase, 1 lowercase, 1 digit, and 1 special
character
- validates username: 3-20 chars, alphanumeric + underscore only returns object { valid:
boolean, errors: string[] }
- uses JSDoc type annotations throughout (or TypeScript-style types if preferred)
- never throws exceptions for validation failures
- is exported as a named ES module export

EXAMPLES:
Input:  { email: "alice@example.com", username: "alice_92",
password: "SecurePass123!" }
Output: { valid: true, errors: [] }

Input:  { email: "not-an-email", username: "ab",
password: "12345" }
```

```
Output: { valid: false, errors: [
"email: Please enter a valid email address.",
"username: Username must be 3-20 characters.",
"password: Password must be at least 12 characters and
include uppercase, lowercase, digit, and special character."
]}
```

What you should get

Production-grade output. Error messages now read like a real product would ship. The function respects the middleware context (no logging of passwords). The I/O shape matches your examples byte-for-byte. You can ship this — the prompt did 90% of the work, you own the 10%. That is the R-I-C-E payoff: five extra minutes of prompt engineering saves thirty minutes of code review.

Remember

The prompt grew from 8 words to 250. The output grew from unusable to shippable. That is not a 30× ratio by accident — context is literally what the model needs to do its job, and you are the only one who has it. Giving context to the model is not “prompt bloat.” It is the job.

Segment 4 — Build the R-I-C-E Analyzer

What to Learn

We just watched prompt quality climb with each R-I-C-E element added. Now we will build a Python CLI that quantifies it — feed it any prompt, get back a score for Role, Instructions, Context, and Examples plus a concrete improvement suggestion for each. This closes the loop: a tool that measures the thing we are teaching.

Step 1. Install the Anthropic SDK.

```
pip install anthropic
also
```
pip install python-dotenv

Step 2. Create rice_analyzer.py with this code.

```
#!/usr/bin/env python3
"""rice_analyzer.py – Score a prompt on the R-I-C-E framework."""
```

from dotenv import load_dotenv
load_dotenv() # must run before Anthropic() is instantiated

```
import argparse, json, os, sys
from anthropic import Anthropic

SYSTEM = """You are a prompt-engineering evaluator.

Score the user prompt on the R-I-C-E framework:
  R = ROLE         — is a persona / expertise assigned to the AI?
  I = INSTRUCTIONS — are tasks, constraints, success criteria explicit?
  C = CONTEXT      — is surrounding business / technical context given?
  E = EXAMPLES     — are input/output examples or references included?

For each dimension, return: score (0-10), present (bool), evidence
(short quote or "N/A"), and improvement (one specific suggestion).
Also return overall_score (0-10) and a one-paragraph summary.

Respond with ONLY valid JSON matching this schema:
{
  "role":         {"score":0,"present":false,"evidence":"","improvement":""},
  "instructions": {"score":0,"present":false,"evidence":"","improvement":""},
  "context":      {"score":0,"present":false,"evidence":"","improvement":""},
  "examples":     {"score":0,"present":false,"evidence":"","improvement":""},
  "overall_score": 0,
  "summary": ""
}
No markdown, no prose — only JSON."""



def analyze(prompt: str) -> dict:
    client = Anthropic()  # reads ANTHROPIC_API_KEY
    msg = client.messages.create(
        model="claude-opus-4-7",
        max_tokens=1024,
        system=SYSTEM,
        messages=[{"role": "user",
                   "content": f"Analyze this prompt:\n\n---\n{prompt}\n---"}],
    )
    raw = msg.content[0].text.strip()
    if raw.startswith("```"):
        raw = raw.split("```")[1]
        if raw.startswith("json"):
            raw = raw[4:]
    return json.loads(raw)
```

```
def render(r: dict) -> str:
    bar = lambda n: "█" * n + "░" * (10 - n)
    out = ["", "=" * 60, "  PROMPT R-I-C-E ANALYSIS", "=" * 60]
    for key, label in [("role", "ROLE        "),
                       ("instructions", "INSTRUCTIONS"),
                       ("context", "CONTEXT     "),
                       ("examples", "EXAMPLES    ")]:
        d = r[key]
        mark = "✓" if d["present"] else "✗"
        out.append(f"  {mark} {label} [{bar(d['score'])}] {d['score']}/10")
        out.append(f"     evidence:    {d['evidence']}")
        out.append(f"     improvement: {d['improvement']}")
        out.append("")
    out.append(f"  OVERALL: {r['overall_score']}/10")
    out.append("  " + "-" * 58)
    out.append(f"  {r['summary']}")
    out.append("=" * 60)
    return "\n".join(out)


def main():
    ap = argparse.ArgumentParser(description="Score a prompt on R-I-C-E.")
    ap.add_argument("prompt", nargs="?")
    ap.add_argument("--file", "-f")
    args = ap.parse_args()

    if args.file:
        prompt = open(args.file, encoding="utf-8").read()
    elif args.prompt:
        prompt = args.prompt
    else:
        prompt = sys.stdin.read()

    if not prompt.strip():
        sys.exit("Error: no prompt provided.")
    if not os.environ.get("ANTHROPIC_API_KEY"):
        sys.exit("Error: set ANTHROPIC_API_KEY.")

    print(render(analyze(prompt)))


if __name__ == "__main__":
    main()
```

Step 3. Run it on Prompt v1 (the naive one):

```
python rice_analyzer.py "Write a validation function for user registration."
```

Step 4. Run it on Prompt v4 (save v4 to prompt_v4.txt first):

```
python rice_analyzer.py --file prompt_v4.txt
```

> **What you should get**
>
> On v1: scores around 1/10, 2/10, 0/10, 0/10 — overall 1/10. The improvement column tells you exactly what to add. Run it on v2, v3, v4 in turn and watch the bars fill up. By v4 you are pegged at 9s and 10s.
>
> You have just built the feedback loop you will use for the next seven modules: write a prompt, score it, add what's missing, re-score. This is how professional prompt engineering becomes a habit instead of an art.

Wrap-Up — The Workshop Is Open

> **Remember**
>
> In 25 minutes you installed two AI-native editors, ran the same prompt through four models, revised a single prompt four times to go from unusable to shippable, and built a tool that will measure your prompts for the rest of this course. Module 1 is not about an app. It is about the fact that you are now operating the cockpit.

Key takeaways:

- VS Code and Cursor are complementary. Keep both. Pick per task.
- Four frontier models, four personalities. Never trust one model for all work.
- R-I-C-E is a grammar, not a checklist. The order Role → Instructions → Context → Examples is how quality compounds.
- Context is measurable. Your Analyzer is the proof.
- Every module from here on assumes these four habits. If anything in today's demo felt shaky, re-run it before Module 2.

> **Student Lab — Before Module 2**

> Pick five real prompts you have written in the last week (Slack questions to an AI, Copilot hints, anything). Paste each into your Analyzer. For every one that scores below 7 overall, rewrite it until it scores 9+. Submit the five before-and-after pairs as Module 1 homework.