2

AI-Augmented Test Design

From Prompts to Test Suites

THE MODERN QA ENGINEER SKILLSET

AI-Augmented Test Design: From Prompts to Test Suites

Prompt Engineering, AI Output Curation, and Strategic Test Design

Yuri Syuganov

Modern QA Course

https://modernQAcourse.com

AI-Augmented Test Design: From Prompts to Test Suites

Prompt Engineering, AI Output Curation, and Strategic Test Design

Book 2 of 26 in the series *The Modern QA Engineer Skillset*

First Edition · Version 1.0.0 · 2026-07-11

Set in Charter, IBM Plex Sans, and IBM Plex Mono. Typeset with Pandoc and XeLaTeX.

Modern QA Course · https://modernQAcourse.com · support@modernQAcourse.com

Contents

Introduction

A Comprehensive Self-Guided Learning Textbook for QA Engineers

Edition: 1.1 | **Last Updated:** July 2026

About This Book

This book teaches you a new way of working. Not a new tool, not a new framework – a new *relationship* with how test suites come into existence.

For decades, QA engineers wrote every test by hand. Line by line, assertion by assertion, edge case by edge case. That era is ending. Large Language Models (LLMs) – Claude, ChatGPT, GitHub Copilot, Cursor – can now generate test code at extraordinary speed. But speed without quality is waste. An AI can produce 50 tests in five minutes, yet half of them may be tautologies that test the mock instead of the code, happy-path-only suites that miss every error condition, or hallucinated API calls to methods that do not exist in your codebase.

The role of the modern QA engineer is not to write every test by hand. It is to **design test strategies**, feed structured context into AI systems, evaluate what comes back, and curate the result into a reliable, maintainable suite. You are promoted from writer to **editor-in-chief**. The AI is a prolific but imprecise junior engineer on your team. Your expertise – in domain knowledge, risk assessment, and engineering judgment – is what transforms raw AI output into production-quality tests.

This book gives you the complete skill set for that role: prompt engineering for test generation, classical test techniques accelerated by AI, sys-

tematic review patterns for AI output, tool-specific workflows, and the strategic judgment to know when to generate and when to write from scratch.

Every concept is accompanied by real code you can run, prompt templates you can copy, exercises you can practice, and decision frameworks you can apply on your next sprint.

Who This Book Is For

This book is designed for:

- **QA Engineers** (manual or automated) who want to integrate AI into their daily workflow
- **SDETs** (Software Development Engineers in Test) looking to accelerate test suite creation
- **Test Leads and Managers** who need to establish quality standards for AI-generated tests
- **Software Engineers** who write their own tests and want to do it faster
- **Career Changers** entering QA who want to learn modern, AI-augmented practices from day one
- **Interview Candidates** preparing to discuss AI-augmented testing in technical interviews

What you should already know:

- Basic programming in at least one language (Python, JavaScript/TypeScript, or Java)
- Familiarity with at least one test framework (pytest, Jest, Vitest, JUnit)
- Understanding of what unit tests, integration tests, and E2E tests are
- Basic understanding of REST APIs and HTTP methods
- Comfort with the command line

What you do NOT need to know:

- Machine learning or how LLMs work internally
- Prompt engineering (we teach this from scratch)
- Any specific AI tool (we cover Claude, ChatGPT, Copilot, and Cursor)

How to Use This Book

Linear Reading Path (Recommended for First Read)

Read Part I through Part V in order. Each part builds on the previous one. The exercises at the end of each chapter reinforce the material before you move on.

Reference Path (For Experienced Practitioners)

If you already use AI for test generation and want to improve specific skills:

- **Prompt quality issues?** Start with Part I (Chapters 1-4)
- **AI output quality issues?** Start with Part III (Chapters 8-11)
- **Tool selection questions?** Start with Part IV (Chapters 12-15)
- **Strategic decisions?** Start with Part V (Chapters 16-19)
- **Testing AI-powered features (LLM evals, RAG)?** Go straight to Chapter 19

Practice Path (For Hands-On Learners)

Each chapter ends with graded exercises. Work through them in order:

- Exercises marked **(Beginner)** can be completed in 10-15 minutes
- Exercises marked **(Intermediate)** require 20-40 minutes
- Exercises marked **(Advanced)** are mini-projects taking 1-2 hours
- The **Capstone Project** in Part VI ties everything together

Conventions Used in This Book

Throughout this book, you will encounter several types of callout boxes:

> **Pro Tip:** Practical advice from real-world experience that goes beyond the textbook material.

> **Common Mistake:** Patterns that look correct but lead to problems. Learn to recognize these before they cost you time.

> **Key Insight:** Conceptual breakthroughs that change how you think about the topic.

> **Interview Note:** How to discuss this concept in a job interview context.

Code blocks labeled with a language (`python`,typescript, “‘bash) contain real, working code. Code blocks without a language label contain prompt templates or pseudocode.

Prompt templates use `{curly_braces}` for placeholders you fill in with your own values.

Table of Contents

Part III: Reviewing and Curating AI Output

- Chapter 8: The 10x Review Pattern
- Chapter 9: The Six Failure Modes of AI-Generated Tests
- Chapter 10: The Mental Mutation Test
- Chapter 11: Quality Gates for CI/CD

Part IV: Tools and Workflows

- Chapter 12: Claude Code – The Spec-to-Suite Workflow
- Chapter 13: GitHub Copilot – Inline Test Completion
- Chapter 14: Cursor – Interactive Test Development
- Chapter 15: The Hybrid Multi-Tool Workflow

Part V: Strategy and Decision-Making

- Chapter 16: The Write vs Generate Decision Matrix
- Chapter 17: The Curation Workflow
- Chapter 18: Economics and Metrics of AI Test Generation
- Chapter 19: LLM Evals and AI Test-Suite Quality

Part VI: Capstone and Reference

- Chapter 20: Capstone Portfolio Project
- Chapter 21: The Complete Prompt Template Library (100 Templates)
- Glossary
- Further Reading

PART I

PROMPT ENGINEERING FOR TEST GENERATION

CHAPTER 1

The Anatomy of a Test-Generation Prompt

1.1 Why Prompt Structure Matters

A poor prompt produces poor tests. This is not a vague principle – it is a direct, measurable relationship. The difference between "write tests for login" and a structured, context-rich prompt is the difference between a junior engineer guessing and a senior engineer analyzing requirements.

LLMs are statistical pattern matchers. They predict the most likely next token given the input tokens. The quality of their output is directly proportional to the specificity and structure of your input. When you write "write tests for login," the LLM retrieves a generic pattern for login testing from its training data. When you write a structured prompt with context, artifacts, constraints, coverage goals, and output format, the LLM has enough signal to generate tests that are specific to your system, your framework, your coding style, and your coverage needs.

In test generation, this means you need to think of your prompt as a **test specification document**, not a casual request. Every piece of context you include (or omit) shapes the coverage, quality, and usefulness of the generated tests.

Consider these two prompts and their typical outputs:

Prompt A: The casual request

```
Write tests for the checkout API.
```

Typical output from Prompt A:

```
test('checkout works', () => {
  const result = checkout({ items: [{ id: 1, qty: 1 }] });
  expect(result).toBeDefined();
});

test('checkout fails with empty cart', () => {
  expect(() => checkout({ items: [] })).toThrow();
});
```

Two tests. Vague assertions. No framework specified. No authentication. No error scenarios. No boundary values. The AI had nothing to work with, so it produced nothing of value.

Prompt B: The structured specification

```
You are a senior QA engineer. Generate a comprehensive test suite for the
checkout API endpoint.

**Technical Context:**
- Backend: Node.js 20, Express 4, TypeScript
- Endpoint: POST /api/v2/checkout
- Auth: JWT Bearer token with "customer" role
- Database: PostgreSQL 15 via Prisma ORM
- Payment: Stripe API (test mode)
- Prices: stored as integers (cents)

**Acceptance Criteria:**
1. Valid cart with in-stock items creates an order with status "pending"
2. Out-of-stock items return 409 with specific item IDs
3. Invalid payment method returns 402
4. Expired JWT returns 401
5. Missing required fields return 400 with field-specific errors

**Constraints:**
- Framework: Vitest + Supertest
- Pattern: Arrange/Act/Assert with comments
- Naming: "should [behavior] when [condition]"
- Mock all external APIs (Stripe, SendGrid)
- Use factory functions from ./tests/factories.ts

**Coverage goals:**
```

```
- All 5 acceptance criteria with at least 1 test each
- At least 2 negative cases per AC
- Boundary values for quantity (0, 1, 100, 101)
- Auth: valid token, expired token, missing token, wrong role

**Output format:**
- Single TypeScript file
- One describe block per AC
- One it block per test case
```

Typical output from Prompt B: 25-35 well-structured tests covering every acceptance criterion, edge case, and boundary value, using the correct framework, following the specified naming convention, and ready to drop into the test directory.

The difference is not subtle. It is the difference between wasted time and productive output.

1.2 The Five Elements of an Effective Test-Generation Prompt

Every high-quality test prompt contains five elements. Missing any one of them degrades output quality significantly. These elements form the acronym **CACOF** (Context, Artifact, Constraints, Output format, … or simply remember them as the "five pillars").

Element	Purpose	What happens if missing
Context	What system/feature is under test	AI makes generic assumptions, hallucinates tech stack
Artifact	The spec, schema, or story to test against	AI invents APIs and fields that do not exist
Constraints	Framework, language, patterns to follow	AI uses wrong framework, wrong style, wrong helpers
Coverage goals	What categories of tests to generate	AI defaults to happy-path-only tests
Output format	How to structure the response	AI produces unstructured, hard-to-integrate code

Let us examine each element in exhaustive detail.

1.3 Element 1: Context

Context tells the LLM what kind of system it is dealing with. Without it, the LLM makes generic assumptions. With it, the LLM tailors its output to the technology stack, domain, and architecture.

What context includes:

1. **Technology stack** – language version, framework, runtime
2. **Architecture** – monolith vs microservices, sync vs async, REST vs GraphQL
3. **Domain** – e-commerce, healthcare, fintech, social media
4. **External dependencies** – payment processors, email services, file storage
5. **Data formats** – prices in cents vs dollars, dates in ISO 8601, UUIDs vs integers
6. **Authentication mechanism** – JWT, OAuth, API keys, session cookies

Weak context vs strong context:

Weak context:

```
Write tests for the checkout flow.
```

The AI does not know:

- What language or framework you use
- What the checkout flow involves
- What external services are called
- How prices are represented
- How authentication works

Strong context:

```
We have an e-commerce checkout API built in Node.js 20 with Express 4.
The checkout flow involves:
1. Cart validation (check stock availability via inventory service)
2. Payment processing via Stripe API (test mode)
3. Order creation in PostgreSQL via Prisma ORM
4. Email confirmation via SendGrid

The API is behind JWT authentication (customer role required).
All prices are stored as integers (cents). A $29.99 item is stored as 2999.
The cart maximum is 50 items. Minimum order value is $1.00 (100 cents).
```

The strong context tells the LLM about the tech stack, the business flow, the external dependencies, the data format, and the auth mechanism. This eliminates entire classes of hallucination.

Context depth by test type:

Test Type	Context Depth Needed	Key Context Items
Unit test	Low	Function signature, input/output types, business rules
Integration test	Medium	Endpoint details, database schema, external service contracts
E2E test	High	Full user flow, UI components, navigation, auth flow
Performance test	Medium	Expected throughput, SLAs, infrastructure details
Security test	High	Auth mechanism, data sensitivity, compliance requirements

> **Pro Tip:** When in doubt about how much context to include, err on the side of more rather than less. You can always reduce context in subsequent iterations if the output is already good. But recovering from missing context requires a full regeneration.

1.4 Element 2: Artifact

The artifact is the primary source of truth for test generation. It is the document that defines what the code *should* do. Without it, the AI generates tests based on common patterns from its training data – which may have nothing to do with your actual system.

Types of artifacts:

Artifact Type	When to Use	How to Feed It
OpenAPI/Swagger schema	API endpoint testing	Paste the relevant YAML/JSON section
User story + acceptance criteria	Feature testing	Copy verbatim from Jira/ Linear
Database schema (DDL)	Data integrity testing	Paste CREATE TABLE statements
Function signature + docstring	Unit testing	Paste the function code
UI mockup description	Frontend testing	Describe layout, or use vision model with screenshot
Error code catalogue	Error handling testing	Paste the error code table
State machine diagram (text)	Workflow testing	Describe states and transitions

Best practice: Paste, do not describe

Pasting the artifact directly into the prompt is far more effective than describing it. Compare:

Describing the artifact (weak):

```
The checkout endpoint accepts a cart_id field (required, UUID format) and a
payment_method_id field (required, string) and an optional coupon_code field.
```

Pasting the artifact (strong):

```
paths:
  /api/v2/checkout:
    post:
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [cart_id, payment_method_id]
              properties:
                cart_id:
                  type: string
                  format: uuid
                payment_method_id:
                  type: string
                coupon_code:
                  type: string
                  pattern: "^[A-Z0-9]{8}$"
      responses:
        '201':
          description: Order created successfully
        '400':
          description: Validation error
        '401':
          description: Unauthorized
        '402':
          description: Payment failed
        '409':
          description: Stock conflict
```

The pasted schema gives the AI exact field names, types, constraints, and response codes. The description is ambiguous and incomplete.

Handling $ref references

When pasting OpenAPI schemas, you must resolve `$ref` references. The AI cannot follow `$ref: '#/components/schemas/Address'` if you do not include the Address schema.

Incomplete (will cause hallucinations):

```
shipping_address:
  $ref: '#/components/schemas/Address'
```

Complete (AI can generate accurate tests):

```
shipping_address:
  $ref: '#/components/schemas/Address'

# Include the referenced schema:
components:
  schemas:
    Address:
      type: object
      required: [street, city, country, postal_code]
      properties:
        street:
          type: string
          maxLength: 200
        city:
          type: string
          maxLength: 100
        country:
          type: string
          pattern: "^[A-Z]{2}$"
        postal_code:
          type: string
          maxLength: 20
```

> **Common Mistake:** Pasting an entire 5000-line OpenAPI spec when you only need 50 lines for one endpoint. This floods the context window with irrelevant information and degrades output quality. Extract only the relevant endpoint and its referenced schemas.

1.5 Element 3: Constraints

Constraints prevent the LLM from generating tests in the wrong framework, language, or style. They also enforce team conventions that make generated code immediately usable.

Essential constraints to specify:

```
**Constraints:**
- Language: TypeScript 5.x
- Framework: Vitest with supertest
- Pattern: Arrange/Act/Assert with explicit comments
- Naming: "should [behavior] when [condition]"
- No external HTTP calls -- mock all Stripe/SendGrid calls
- Use factory functions from ./tests/factories.ts
- Each test file must import { describe, it, expect } from 'vitest'
- Use beforeEach for test isolation, not beforeAll
- Maximum 25 lines per test function
- No console.log statements in test code
```

The style reference constraint

One of the most powerful constraints is a style reference: 2-3 existing tests from your codebase that demonstrate the expected patterns.

````
**Style reference (existing test from our codebase):**
```python
class TestUserCreation:
    def test_should_create_user_when_valid_email(self, db_session, user_factory):
        # Arrange
        dto = user_factory.build(email="new@example.com")

        # Act
        user = UserService(db_session).create_user(dto)

        # Assert
        assert user.id is not None
        assert user.email == "new@example.com"
        assert user.created_at is not None
````

Follow this exact pattern for all generated tests.

````
When the AI sees concrete examples, it replicates:

- Class structure with docstrings
- Fixture-based dependency injection (`db_session`, `user_factory`)
- Naming convention: `test_should_X_when_Y`
- Comment markers for Arrange/Act/Assert
- Assertion style (check specific fields, not just status codes)
- Helper usage patterns

> **Pro Tip:** Provide one happy-path test and one error-handling test as style references. This teaches the AI both patterns in one
↪  shot.

## 1.6 Element 4: Coverage Goals

Without explicit coverage goals, the LLM defaults to happy-path tests. This is perhaps the most critical element to get right,
↪  because it directly determines whether your test suite catches bugs or merely occupies disk space.

### Explicit coverage requirements:
````
````

Coverage requirements: - Every acceptance criterion must have at least one test - Include at least 2 negative/error cases per endpoint - Include 1 boundary value test for every numeric field - Test auth: valid token, expired token, missing token, wrong role - Test idempotency: calling the endpoint twice with the same cart_id - Test concurrent access: two users modifying the same resource - Test pagination: first page, last page, empty page, page beyond total

```
### Coverage categories to consider:

| Category | Examples | Why AI Misses It |
|----------|----------|------------------|
| Happy path | Valid input, successful response | AI covers this well |
| Validation errors | Missing fields, wrong types, out-of-range | AI covers partially |
| Auth failures | No token, expired token, wrong role | AI often omits role-based tests |
| Resource conflicts | Duplicate key, stock conflict, race condition | AI rarely considers |
| Boundary values | Min, max, min-1, max+1 for every numeric field | AI covers if prompted |
| State transitions | Valid and invalid state changes | AI almost never considers |
| Pagination | Edge cases around page boundaries | AI rarely generates |
| Idempotency | Repeat requests, duplicate submissions | AI almost never considers |
| Concurrency | Parallel modifications, optimistic locking | AI cannot generate well |

### The negative-to-positive ratio

A good test suite has approximately a 2:1 ratio of negative (error) tests to positive (success) tests. This is because there are
↪  more ways for things to go wrong than right. If the AI generates 10 tests and 8 are happy-path, your prompt needs stronger
↪  coverage goals.
```

Coverage ratio guidance: For each endpoint: - 2-3 happy path tests (valid variations) - 4-6 validation error tests (missing fields, wrong types, boundaries) - 2-3 auth error tests (no token, wrong role, expired) - 1-2 conflict/state tests (duplicate, out-of-stock) - 1-2 edge case tests (empty arrays, maximum sizes, special characters)

```
## 1.7 Element 5: Output Format

Controlling the output format makes the generated code immediately usable rather than requiring reformatting.

### Format specifications:
```

Output format: - One describe block per endpoint - One it block per test case - Group related tests with nested describe blocks - Include JSDoc comment above each test explaining what it verifies - Output as a single TypeScript file ready to save to tests/ - Tests should be ordered: happy path first, then errors, then edge cases - Use consistent indentation (2 spaces) - Include all necessary imports at the top of the file

```
### Format for different outputs:

#### When you want code:
```

Output as a single Python file with pytest fixtures at the top.

```
#### When you want a test plan:
```

Output as a numbered markdown table with columns: Test ID, Scenario, Input, Expected Result, Priority, Type (unit/integration/e2e)

```
#### When you want BVA analysis:
```

Output as a markdown table with columns: Field, Input Value, Expected (accept/reject), Rationale

```
> **Common Mistake:** Not specifying the output format and getting a mix of explanation text and code blocks that requires manual
↪  extraction. Always tell the AI exactly what shape you want the output in.

## 1.8 Putting It All Together: A Complete Prompt

Here is a complete prompt that includes all five elements:
```

You are a senior QA engineer. Given the following user story, acceptance criteria, and technical context, generate a comprehensive test suite.

User Story: As a customer, I want to apply a coupon code during checkout so that I get a discount on my order.

Acceptance Criteria: 1. Valid coupon codes reduce the order total by the specified percentage 2. Expired coupons return a clear error message 3. Coupons can only be used once per customer 4. Invalid coupon format is rejected before hitting the database

Technical Context: - Backend: Node.js 20, Express 4, TypeScript - Database: PostgreSQL 15 via Prisma ORM - Test framework: Vitest + Supertest - Auth: JWT tokens via get_test_token("customer") helper - Coupon format: 8 uppercase alphanumeric characters (regex: [1]{8}$)

Generate: 1. Unit tests for coupon validation logic (no DB, no HTTP) 2. Integration tests for the POST /api/v2/checkout endpoint with coupon 3. Edge case tests (empty string coupon, SQL injection in coupon field, etc.)

For each test, include: - Test name: "should [expected behavior] when [condition]" - Arrange/Act/Assert structure with comments - Explicit assertions (not just "expect something")

Coverage requirements: - All 4 acceptance criteria must have at least one test - At least 2 negative/error cases per AC - Boundary test for coupon format (7 chars, 8 chars, 9 chars)

[1] A-Z0-9

```
This prompt contains all five elements:
1. **Context:** Node.js 20, Express 4, TypeScript, PostgreSQL, Prisma
2. **Artifact:** User story with 4 acceptance criteria
3. **Constraints:** Vitest + Supertest, AAA pattern, naming convention
4. **Coverage goals:** All ACs, 2 negatives per AC, boundary tests
5. **Output format:** Test name pattern, AAA structure, explicit assertions

## 1.9 The Prompt Iteration Loop

Prompt engineering is not one-shot. The first prompt rarely produces perfect output. Use this iteration loop:
```

1. DRAFT the prompt with all five elements
2. GENERATE a small batch (5-10 tests)
3. EVALUATE: Are assertions correct? Are APIs real? Is the style right?
4. REFINE the prompt based on what was wrong
5. REGENERATE with the improved prompt
6. REPEAT until output quality is consistently high
7. SAVE the refined prompt as a template for future use

```
This loop typically converges in 2-3 iterations. The investment in prompt refinement pays off every time you reuse the template for
↪  a similar feature.

### Iteration example:

**Iteration 1 output problem:** AI used `jest` instead of `vitest`
**Prompt fix:** Added explicit import statement to constraints

**Iteration 2 output problem:** AI generated only happy-path tests
**Prompt fix:** Added explicit negative-to-positive ratio guidance

**Iteration 3 output:** Quality meets standards. Save as template.

## 1.10 Self-Assessment Quiz

Test your understanding of Chapter 1:

1. What are the five elements of an effective test-generation prompt?
2. Why is pasting an OpenAPI schema more effective than describing it?
3. What is the typical negative-to-positive test ratio in a well-designed suite?
4. Why should you provide a style reference in your prompt?
5. How many iterations does the prompt refinement loop typically require?
6. What happens when you ask for 100 tests in a single prompt?
7. Name three things that strong context tells the LLM that weak context does not.

#### Answers:
1. Context, Artifact, Constraints, Coverage Goals, Output Format
2. The schema provides exact field names, types, constraints, and response codes -- eliminating ambiguity and hallucination
3. Approximately 2:1 negative to positive
4. The AI replicates naming conventions, fixture usage, assertion style, and code structure from examples
5. 2-3 iterations
6. Quality degrades in later tests; generate in batches of 10-20 instead
7. Technology stack details, external dependencies, data formats, authentication mechanism, business flow, architecture style

## 1.11 Key Takeaways

- Treat your prompt as a test specification document, not a casual request
- The five elements (Context, Artifact, Constraints, Coverage Goals, Output Format) form a complete test generation specification
- Missing any single element degrades output quality significantly
- Paste artifacts directly rather than describing them
- Include 2-3 existing tests as a style reference
- Explicitly request negative cases, boundary values, and auth scenarios
- Iterate on your prompts and save refined versions as templates
- Generate in batches of 10-20 tests, not 100 at once

## 1.12 Exercises

**Exercise 1.1 (Beginner):** Take a feature you recently tested at work. Write a test-generation prompt that includes all five
↪  elements. Do not run it yet -- just write the prompt. Score yourself: does it include Context? Artifact? Constraints? Coverage
↪  Goals? Output Format?

**Exercise 1.2 (Beginner):** Compare these two prompts. List every problem with Prompt A and identify which element is missing or
↪  weak:
```

```
Prompt A: "Write tests for user registration."

Prompt B: "Generate pytest tests for the POST /api/v1/users endpoint. The endpoint accepts {email, password, name} with email
↪  requiring valid format, password minimum 8 characters with at least one digit. Use the existing test pattern from test_auth.py.
↪  Include tests for valid registration, duplicate email, short password, invalid email format, and missing required fields."

**Exercise 1.3 (Intermediate):** Find an OpenAPI schema (your project's or a public one like Petstore). Extract one endpoint and
↪  write a complete five-element prompt for it. Feed it to an AI tool and evaluate the output: how many tests were generated? How
↪  many are immediately usable? What would you change in the prompt?

**Exercise 1.4 (Advanced):** Using the prompt iteration loop, refine a prompt through 3 iterations. Document each iteration: what
↪  was wrong, what you changed, and how the output improved. Save the final prompt as a template.

## Career Translation

### Resume phrasing

- Engineered structured test-generation prompts using the five-element framework (Context, Artifact, Constraints, Coverage Goals,
↪  Output Format), transforming AI tools from generic code generators into precision test suite producers
- Reduced AI-generated test rework by 50% through systematic prompt iteration, achieving 25-35 well-structured tests per prompt with
↪  correct framework, naming conventions, and coverage targets
- Established prompt-as-specification practice within QA team, treating each prompt as a test specification document with measurable
↪  impact on test generation quality

### Cover letter framing
I treat AI prompt engineering as a core QA engineering skill, not a casual activity. By structuring every test-generation prompt
↪  with five essential elements -- technical context, specification artifacts, framework constraints, explicit coverage goals, and
↪  output format requirements -- I consistently produce AI output that is immediately usable in production test suites. This
↪  systematic approach transforms AI from a novelty tool into a reliable test generation engine, producing 25-35 targeted tests per
↪  prompt versus the typical 2-3 generic tests from unstructured requests.

### Interview framing
"I approach AI test generation by treating the prompt as a test specification document. Every prompt I write contains five elements:
↪  context about the system and tech stack to prevent hallucination, the actual specification artifact pasted directly rather than
↪  described, explicit constraints on framework and coding patterns, measurable coverage goals with negative-to-positive ratios,
↪  and output format specifications. I iterate through 2-3 refinement cycles, tracking what went wrong and why. The result is a
↪  reusable template that the team can apply to similar features. Trade-offs include the upfront investment in prompt engineering
↪  versus the massive payoff in test quality and reusability."

### What not to say

- "I just ask the AI to write some tests" -- vague prompts produce vague tests; this signals no prompt engineering skill
- "Prompt engineering is not a real skill" -- it is the primary differentiator between useful and useless AI output for test
↪  generation
- "I write one prompt and expect perfect output" -- prompt refinement through iteration is essential and normal
- "More words in the prompt always means better output" -- bloated prompts with irrelevant context degrade quality
- "I do not need to specify the test framework" -- without it, the AI guesses and often guesses wrong

## Interview Depth Check

### Question 1
**Prompt:** You are asked to generate tests for a checkout API at a new company. You have never seen the codebase. What information
↪  do you gather before writing your first prompt?

#### What a strong answer should cover:

- All five elements mapped to information sources
- Prioritizing specification and existing test patterns
- Identifying the tech stack, auth mechanism, and data formats
- Asking about team conventions and anti-patterns

#### Example answer:

- Context: I ask for the tech stack (language, framework, database, runtime version), architecture style (monolith vs
↪  microservices), and external service dependencies (payment processor, email service)
- Artifact: I request the OpenAPI schema or Swagger file for the checkout endpoint, plus any acceptance criteria or user stories
↪  from the project management tool
- Constraints: I ask to see 2-3 existing test files to understand naming conventions, fixture patterns, assertion style, and which
↪  testing libraries are used. I also ask about team anti-patterns -- "What are the things you do not want in tests?"
- Coverage Goals: I ask what the team considers high-priority: is it coverage breadth, error handling depth, or performance
↪  scenarios? What is the target negative-to-positive test ratio?
- Output Format: I ask how tests are organized (one file per endpoint? per feature?), what grouping structure is used (describe
↪  blocks? test classes?), and whether tests should include inline comments

### Question 2
**Prompt:** You send a prompt and get back 10 tests. Seven are happy-path tests, two have vague assertions, and one references a
↪  method that does not exist. Diagnose the prompt issues.

#### What a strong answer should cover:

- Happy-path bias indicates missing coverage goals for negative/error cases
- Vague assertions indicate missing output format specifications
- Hallucinated method indicates missing or incomplete artifact
- Specific prompt fixes for each issue

#### Example answer:

- Seven happy-path tests out of 10: my coverage goals did not specify a negative-to-positive ratio or list specific error scenarios.
↪  Fix: add "Include at least 2 negative/error cases per acceptance criterion" and explicitly list the error conditions (missing
↪  fields, wrong types, auth failures)
```

```
- Vague assertions like "expect(result).toBeDefined()": my output format constraints did not specify assertion depth. Fix: add
↪  "Every test must assert on specific field values, not just existence or status codes"
- Hallucinated method: my artifact was incomplete -- either I described the API instead of pasting the schema, or I pasted the
↪  schema without resolving $ref references. Fix: paste the actual endpoint schema with all referenced components resolved
- After making these three fixes, I regenerate and expect a much better result. If the second iteration still has issues, I refine
↪  further and save the final version as a template

### Question 3
**Prompt:** How do you measure whether a test-generation prompt is "good"?

#### What a strong answer should cover:

- Quantitative metrics: first-run pass rate, deletion rate, negative-to-positive ratio
- Qualitative metrics: assertion quality, style match, coverage completeness
- Comparison against the five-element checklist
- Iteration tracking across refinement cycles

#### Example answer:

- First-run pass rate: what percentage of generated tests compile and pass without modification? Target: 70-85%. Below 60% means the
↪  context or constraints are wrong
- Deletion rate: what percentage of tests are deleted during review as tautologies, duplicates, or irrelevant? Target: 10-20%. Above
↪  30% means the coverage goals or artifact are insufficient
- Negative-to-positive ratio: target 2:1. If the ratio is 1:4, the coverage goals need explicit negative case requirements
- Style match: do the generated tests use the correct fixtures, naming convention, and assertion patterns? If not, the constraints
↪  or style reference need improvement
- Five-element checklist: I score each prompt element 1-3 (weak/adequate/strong) and target a total of 12+/15. Missing elements are
↪  the most common root cause of poor output
- I track these metrics across iterations: if iteration 2 improves pass rate from 60% to 80%, the prompt change was effective

### Question 4
**Prompt:** A junior engineer on your team complains that AI-generated tests are useless because "the AI just makes stuff up." How
↪  do you respond?

#### What a strong answer should cover:

- Validating the observation while identifying the root cause
- Explaining that hallucination is a prompt quality issue, not an AI limitation
- Demonstrating the difference between an unstructured and structured prompt
- Offering to help the engineer improve their prompts

#### Example answer:

- The observation is valid -- with poor prompts, AI absolutely generates hallucinated tests. The root cause is not that AI is
↪  useless but that the prompt lacked the right information
- I would show the comparison: "Write tests for checkout" produces 2 generic, hallucinated tests. A structured prompt with the
↪  OpenAPI schema, existing test patterns, and explicit coverage goals produces 25-35 specific, accurate tests
- The key insight: AI hallucination in test generation is directly proportional to missing context. If the AI invents a method name,
↪  it is because you did not provide the real method names. If it uses the wrong framework, you did not specify the right one
- I would offer to pair with the engineer on their next test generation task, walking through the five-element framework together
↪  and showing the before/after quality difference
- The goal is to transform their perspective from "AI is unreliable" to "AI output quality is proportional to prompt quality, which
↪  I can control"

---


```{=latex}
\cleardoublepage
\backmatter
```
```

About This Sample

You have just read **Chapter 1 of AI-Augmented Test Design: From Prompts to Test Suites** — the free sample. The complete book continues with every remaining chapter, the interview Q&A, and all of the code.

Get the full book

Sample readers get a discount at
https://leanpub.com/modernqa-02-ai-augmented-test-design/c/SAMPLE10

Read online first

Chapter one of **every** book in the series is free to read at
https://modernqacourse.com/library

About the Author

Yuri Syuganov is a software quality engineer with more than twenty years of hands-on testing experience, spanning the arc from manual test design through Selenium-era automation to today's AI-augmented, agent-driven frameworks. He is the author of The Modern QA Engineer Skillset, a 26-book series built on a blunt analysis of what companies actually expect from QA engineers, and the creator of modernQAcourse.com, where the course behind the series is free and readers can bring real testing problems to experienced practitioners. He revises the series as the industry changes — because it does.

Also in This Series

The Modern QA Engineer Skillset — a 26-book course in modern software quality assurance.

AI-Augmented QA (Books 1–10)

- **Book 1:** Agent Skills for Browser Automation: The Complete Guide
- **Book 2:** AI-Augmented Test Design: From Prompts to Test Suites
- **Book 3:** Agentic Testing Architectures: Building Autonomous QA Systems
- **Book 4:** API and Contract Testing with AI: Schema to Shield
- **Book 5:** Performance and Chaos Engineering: Breaking Things on Purpose
- **Book 6:** Observability-Driven Testing: Testing in Production
- **Book 7:** Security Testing for AI Applications: The New Attack Surface
- **Book 8:** Infrastructure as Code Testing: Terraform to Kubernetes
- **Book 9:** Mobile and Cross-Platform Testing: 24,000 Devices, One Strategy
- **Book 10:** Visual and Accessibility Testing: Pixels and People

Technical Foundations (Books 11–19)

- **Book 11:** Manual Testing Fundamentals: The Bedrock
- **Book 12:** Programming for QA: Python, JavaScript, and TypeScript from Zero to Fluent
- **Book 13:** Browser Test Automation: Playwright, Patterns, and the Death of Fragile Tests

- **Book 14:** API Testing Fundamentals: Postman to Pytest
- **Book 15:** SQL and Database Testing: Query Your Way to Quality
- **Book 16:** CI/CD Pipelines: From Commit to Confidence
- **Book 17:** Git and Version Control: Beyond Push and Pull
- **Book 18:** Test Management Tools: Making Quality Visible
- **Book 19:** Linux and Command Line: Your Production Survival Kit

Professional Skills (Books 20–26)

- **Book 20:** Agile and Scrum for QA: Quality in Every Sprint
- **Book 21:** Communication and Stakeholder Management for QA: Influence Without Authority
- **Book 22:** Test Strategy and Quality Metrics: Measuring What Matters
- **Book 23:** QA Leadership and Mentoring: Multiplying Your Impact
- **Book 24:** Technical Writing for QA: Documents That Drive Action
- **Book 25:** Interview Preparation and Career Strategy: Land the Job
- **Book 26:** Testing Like a Senior: Patterns & Anti-Patterns Across Software QA Career Levels

Visit https://modernQAcourse.com for the full course catalog, updates, and companion resources.

Stuck on a Real Problem?

The full course behind this series is free to read at https://modernQAcourse.com — and it stays free.

When a chapter meets the real world and something breaks, post it on the **Help Board** at https://modernQAcourse.com. Asking is free, and experienced practitioners answer right in the thread. Need hands-on help? Book a 1:1 screen-share session with a QA expert.