// pankajnakhat.com · 2026

Prompt Engineering for Quality Engineering

A practitioner's hands-on guide to AI-augmented testing.
50+ production-tested prompts. Real patterns from enterprise QE.

```
pankaj ~ guide.ts
─────────────────────────────────────
$ cat about.ts

const guide = {
author: "Pankaj Nakhat",
role: "Head of QA, Release & Reliability",
location: "Abu Dhabi, UAE",
chapters: 16,
prompts: "50+",
teams: "25+",
experience: "22 years",
};
$ _
```

AI Testing	Prompt Engineering	LLMs	Playwright	BDD	Multi-Agent

Pankaj Nakhat | linkedin.com/in/pankajnakhat

// Contents

// 00

Preface

Quality Engineering is going through its biggest shift since Agile. AI and large language models have moved from "interesting experiment" to "serious part of the engineering toolkit" faster than most of us expected. But here's what bothers me: most QA teams are still using AI the same way they used Google in 2010. Vague question in. Generic answer out. Nothing changes.

That's not because the tools aren't capable. It's because we haven't developed the muscle for prompting them well in a QE context. The practitioner literature is thin, fragmented, and mostly written by people who have never had to triage a failing test suite at 2 AM the night before a production release.

This guide is different. Everything in here comes from real work: leading quality, release, and reliability across 25+ product teams in a Spotify-aligned engineering organisation, building multi-agent test automation systems, integrating LLMs into CI/CD pipelines, and evaluating AI outputs against real production defects. The examples are real. The anti-patterns are real. The cost considerations are real.

By the time you finish, you'll have a solid mental model for prompt engineering as a Quality Engineering discipline, with techniques you can apply immediately across test generation, API validation, performance testing, security probing, LLM evaluation, and multi-agent orchestration.

Who this guide is for

QA Engineers, SDETs, QA Leads, and Quality Engineering Managers who want to systematically integrate prompt engineering into their day-to-day work. You need a working knowledge of software testing. You don't need any prior AI or ML background.

// 01

Foundations of Prompt Engineering

1.1 What prompt engineering actually is

A prompt is the instruction you send to a large language model. Prompt engineering is the discipline of crafting those instructions to reliably get high-quality, useful output. It sits at the intersection of natural language, domain knowledge, and systems thinking.

Here's a framing that clicks for most QE people: think of it as test design for AI systems. A poorly written test case produces ambiguous results. A poorly constructed prompt produces ambiguous, incomplete, or hallucinated outputs. Same problem, different medium.

1.2 The anatomy of a prompt

Every effective QE prompt contains some combination of these elements. You don't need all seven every time, but knowing each one lets you diagnose why a prompt is failing.

Element	What it does	QE example
Role / Persona	Sets the model's context and expertise	"You are a senior SDET specialising in REST API contract testing..."
Context	Background the model needs	OpenAPI spec, user story, existing test code
Task	The specific instruction	"Generate Playwright test cases covering all acceptance criteria..."
Format	Structure of the expected output	"Output as TypeScript using Page Object Model, with AAA comments"
Constraints	Boundaries the model must respect	"Do not use data-testid selectors. Use ARIA roles only."
Examples	One or more input/output demos	A sample test showing the exact structure you want
Evaluation criteria	How output will be judged	"Each test must have exactly one assertion per logical expectation"

1.3 Core prompting strategies

Zero-shot prompting

Asking the model to do something with no examples. Works fine for simple tasks. The quality ceiling is lower than few-shot for complex, domain-specific QE work.

```
Generate three negative test cases for a REST endpoint
that accepts a user registration payload.
```

Few-shot prompting

Providing one or more input/output examples before the task. This is the single highest-leverage technique for QE work because it lets you encode your team's coding standards and test structure patterns directly in the prompt. If I could only recommend one technique, this would be it.

```
Example input:  POST /users with missing email field
Example output:
test('should return 400 when email is missing', async ({ request }) => {
  const response = await request.post('/users', { data: { name: 'Test User' } });
  expect(response.status()).toBe(400);
  const body = await response.json();
  expect(body.error).toContain('email');
});

Now generate: POST /users with email format invalid (no @ symbol)
```

Chain-of-thought (CoT) prompting

Telling the model to reason step by step before producing output. Essential for complex test scenario design where intermediate reasoning produces much better outcomes than jumping straight to generation.

```
Before generating test cases, first:
1. Identify all input parameters and their types
2. Apply boundary value analysis to each numeric field
3. List all error states the API can return
4. Identify state-dependent scenarios

Then generate test cases covering all identified scenarios.
```

Role prompting

Giving the model a specific expert persona. A model prompted as a "penetration tester" probes for different vulnerabilities than one prompted as an "API contract testing specialist." Particularly effective in QE because different testing disciplines need different cognitive frameworks.

1.4 Model behaviours every QE must understand

Behaviour	What it means for your prompts
Recency bias	Put your most important constraint at the END of the prompt, not just the beginning.
Sycophancy	The model will agree with incorrect assertions. Prompt it to find problems, not confirm assumptions.
Hallucination	Models confidently fabricate API methods and test IDs. Every generated test must be executed before you trust it.

Behaviour	What it means for your prompts
Token limits	Long prompts cost more and degrade quality. Inject only the relevant section of a spec, not the whole file.
Instruction following	"Must include" outperforms "should include" every time. Be explicit.

// 02

Prompt Patterns for Test Case Generation

2.1 The test generation meta-pattern

Effective test generation prompts follow a consistent structure. Think of it as a contract between you and the LLM:

```
ROLE: You are a senior QA Engineer specialising in [DOMAIN].

CONTEXT:
[USER STORY / ACCEPTANCE CRITERIA]
[RELEVANT CODE / API CONTRACT]

TASK:
Generate [N] test cases covering [COVERAGE STRATEGY].

CONSTRAINTS:
- Framework: [Playwright / Jest / WebdriverIO]
- Language: [TypeScript / JavaScript / Python]
- Pattern: [Page Object Model / Screenplay]
- [Any team-specific rules]

FORMAT:
[Concrete example of the expected output structure]

EVALUATION:
For each test, explain in a comment why it was included.
```

2.2 Acceptance criteria to test case conversion

One of the highest-ROI applications of prompt engineering in QE. The key is giving the model enough structural context to understand your testing architecture, not just the requirements.

```
Given the following user story and acceptance criteria:
[INSERT AC]

And given our existing Page Object:
[INSERT POM CLASS]

Generate Playwright TypeScript tests covering every Given/When/Then scenario.
Each test must be independent, use the provided POM, and follow AAA pattern.
Flag any acceptance criterion that is ambiguous or untestable.
```

2.3 Boundary value analysis prompting

BVA is systematic but tedious manually. LLMs are genuinely good at this when given structured input.

```
Perform boundary value analysis on the following field specs
and generate a test for each boundary:

Field: age (integer, min: 18, max: 120, required: true)
Field: username (string, minLength: 3, maxLength: 50)
Field: discountCode (string, optional, length: exactly 8 chars)

For each field generate tests for:
- Minimum valid value
- Maximum valid value
- One below minimum (invalid)
- One above maximum (invalid)
- Null/missing value / Type mismatch

Output as Jest tests: 'should [OUTCOME] when [FIELD] is [VALUE]'
```

2.4 Negative testing and error path generation

QA teams consistently under-test error paths because they require more creativity than happy-path tests. Prompting specifically for negative scenarios closes this gap reliably.

```
You are a hostile user trying to break this API endpoint:
[INSERT ENDPOINT SPEC]

Generate 15 negative test cases covering:
- Invalid data types for each field
- Boundary violations and missing required fields
- Malformed JSON payloads
- SQL injection and XSS payload patterns
- Race condition scenarios
- Authentication bypass attempts

For each test: explain the attack vector and expected error response.
```

2.5 Regression test prioritisation

```
Given these code changes from our pull request diff:
[INSERT DIFF]

And our existing test suite inventory:
[INSERT TEST LIST]

1. Identify tests that directly cover changed code paths
2. Identify tests covering code transitively dependent on changes
3. Recommend a prioritised smoke suite (top 20% by risk coverage)
4. Flag changed code paths not covered by existing tests

Output as JSON: { critical_tests[], smoke_suite[], coverage_gaps[] }
```

// 03

API Testing with Prompt Engineering

3.1 OpenAPI spec-driven test generation

The OpenAPI specification is the richest context source for API test generation. The challenge is token budget: extract only the endpoint you need, not the whole file.

```
Extract from your OpenAPI spec:
PATH: POST /api/v2/orders
REQUEST: [INSERT SCHEMA]
RESPONSES: [200, 400, 401, 403, 422, 500 definitions]
SECURITY: bearerAuth

Generate a complete Supertest/Jest test suite that:
1. Tests all documented response codes
2. Validates response schema against the OpenAPI definition
3. Tests auth: valid token, expired token, missing token
4. Includes Zod contract validation derived from the spec
5. Uses our helper: import { getAuthToken } from '../helpers/auth'
```

3.2 Authentication flow testing

```
Context: Our API uses OAuth2 client_credentials for service-to-service
and authorization_code for user-facing calls.
Token endpoint: [INSERT URL]

Generate a TypeScript auth helper module that:
1. Fetches tokens for both grant types
2. Caches tokens with 60-second pre-expiry refresh buffer
3. Handles 500 errors with exponential backoff (max 3 retries)
4. Supports test isolation (separate token pools per test worker)

Export: getServiceToken(), getUserToken(userId), clearTokenCache()
```

3.3 Schema validation prompts

Validation type	Prompt instruction
Type validation	Validate all fields match their OpenAPI types. Test type coercion edge cases (string '123' vs integer 123).
Null handling	Generate tests for every optional field with null, undefined, empty string, and zero values.
Enum validation	Test all valid enum values, one invalid value, and case-sensitivity behaviour.

Validation type	Prompt instruction
Nested objects	Test each nested object: missing entirely vs present with required fields missing.
Array fields	Test arrays with: empty, single item, max+1, items with invalid types.
Date/time formats	Test ISO 8601 valid, Unix timestamp (should fail), future/past date boundaries.

// 04

UI and End-to-End Test Generation

4.1 The UI testing challenge

UI test generation is harder than API testing because it requires understanding visual hierarchy, interaction patterns, and DOM structure. The solution is maximum structural context: DOM snapshots, component docs, or accessibility trees. Don't ask the model to guess what your UI looks like.

4.2 Page Object Model generation

```
Given this React component: [INSERT COMPONENT CODE]

Generate a Playwright TypeScript Page Object Model class that:
1. Imports from '@playwright/test'
2. Uses ARIA roles and accessible names as primary locator strategy
   (no data-testid unless aria role is unavailable)
3. Exposes methods for each user interaction, not raw locators
4. Includes JSDoc comments for each method
5. Follows naming convention: [ComponentName]Page

DO use: getByRole(), getByLabel(), getByText()
DO NOT: CSS selectors, XPath, nth-child
```

4.3 User journey to E2E conversion

```
Given this user journey for [FEATURE]: [INSERT STEPS].
Given these Page Objects: [INSERT POM CLASSES].

Generate a Playwright E2E test that:
(1) Sets up test data via API
(2) Executes each step using the provided POMs
(3) Asserts expected state after each critical step, not just the final state
(4) Tears down test data via API
(5) Is tagged @e2e and @[FEATURE] for filtering
```

4.4 Accessibility test generation

```
You are a WCAG 2.1 Level AA accessibility testing specialist.
Given this component: [INSERT COMPONENT]

Generate Playwright tests checking:
1. Keyboard navigation: all interactive elements reachable via Tab
2. Focus indicators: visible focus ring on all focusable elements
3. ARIA labels: all form inputs have associated labels
4. Color contrast: flag elements that may fail 4.5:1 ratio
5. Screen reader: heading hierarchy is logical (no skipped levels)
```

```
Use @axe-core/playwright for automated checks.
Supplement with manual check instructions in comments.
```

// 05

Performance and Security Testing

5.1 Performance test design

```
Context: Our checkout API serves [N] monthly active users.
Peak load: weekdays 09:00-11:00 local time.
P95 SLA: 800ms. Infrastructure: 3x Node.js serverless behind API gateway.

Design a k6 performance test strategy covering:
1. Baseline: 10 users, 5 min -- establish current P50/P95/P99
2. Load: ramp 0 to [PEAK] users over 10 min, hold 20 min
3. Stress: exceed peak by 50% to find breaking point
4. Spike: 10x sudden traffic for 1 min
5. Soak: 80% load for 4 hours (memory leak detection)

For each type: VU count, duration, thresholds, key metrics to monitor.
```

5.2 OWASP Top 10 coverage prompts

> **Scope note**
>
> The following prompt patterns are designed for authorised security testing on systems you own or have written permission to test. Always operate within your organisation's penetration testing policies.

```
You are an application security tester.
Given our Node.js API with OAuth2 authentication, generate test cases
covering the OWASP API Security Top 10:

1. Broken Object Level Authorization
   -- Tests accessing other users' resources by manipulating IDs
2. Broken Authentication
   -- Test expired tokens, malformed JWTs, missing Bearer prefix
3. Broken Object Property Level Authorization
   -- Test mass assignment: send extra fields in request body
4. Unrestricted Resource Consumption
   -- Test missing rate limits on resource-intensive endpoints
5. Injection
   -- SQLi and NoSQLi payloads for all string input fields

For each: expected HTTP status, expected error pattern, remediation.
```

// 06

Multi-Agent QA Architectures

6.1 Why multi-agent for QA?

Single-agent LLM interactions are limited by context window size, inability to parallelise, and lack of specialisation. Multi-agent architectures solve this by decomposing the QA workflow into specialised, collaborating agents. The five-agent architecture below has been tested across enterprise QA workflows.

Agent	Responsibility	Key prompting pattern
Analyst	Reads requirements. Produces structured test plan JSON.	Chain-of-thought: reason through coverage gaps before outputting.
Writer/Engineer	Consumes the test plan and generates executable test code.	Few-shot: provide 2-3 example tests in your exact team coding style.
Executor/Sentinel	Runs tests, captures results, classifies failures.	Role: "Distinguish flaky tests from genuine regressions."
Healer	Analyses broken selectors. Proposes minimal-diff fixes.	Constrained: "Output only changed lines as a unified diff. Do not refactor."
Oracle	Synthesises results into executive quality reports.	Structured output: respond only with JSON conforming to this schema.

6.2 Prompting the Analyst Agent

```
You are a senior QA analyst. Analyse the following user story and ACs:
[INSERT STORY & AC]

Produce a structured test plan as a JSON object:
{
  "feature": string,
  "riskLevel": "HIGH" | "MEDIUM" | "LOW",
  "scenarios": [{
    "id": string,
    "title": string,
    "type": "happy_path"|"negative"|"boundary"|"security",
    "steps": string[],
    "expectedResult": string
  }],
  "coverageGaps": string[]
}

Do not output anything except the JSON object.
```

6.3 Approval gates

Spec Review Gate: Analyst output reviewed before Writer runs

Diff Review Gate: Healer fixes reviewed before merge

Data Gate: Any test creating/modifying data requires human approval

Commit Gate: Generated tests go to feature branch, not main, pending PR review

// 07

LLM Evaluation and Prompt Regression Testing

7.1 Why QE teams must own LLM evaluation

When your organisation ships AI-powered features, the QE team must own the quality gate for LLM outputs. Hallucination, relevance drift, and toxicity are production defects. Treat the LLM as a system under test (SUT), its prompt as the input, and its generated text as the output to assert against. This is not a new discipline. It's the same job, new medium.

7.2 DeepEval metrics for QA teams

Metric	What it measures	QE use case
Answer Relevancy	Does the output answer the question?	Test that a support bot answers domain queries, not off-topic content
Faithfulness	Does the output contain only claims from context?	RAG systems: verify no facts added beyond retrieved documents
Contextual Precision	Is retrieved context relevant to the query?	Validate vector database retrieval quality
Hallucination	Does the output contradict the source context?	Legal/compliance features where fabrication is a regulatory risk
Toxicity	Does the output contain harmful content?	User-facing AI exposed to public input

7.3 Adversarial prompt testing

Five vectors to test against every AI feature you ship:

- Prompt injection: embed instructions in user input to override system prompt
- Jailbreaking: use roleplay or hypothetical framings to extract restricted content
- Data extraction: probe whether the model reveals training data or context
- Scope violation: ask questions outside the intended domain to test guardrails
- Indirect injection: embed malicious instructions in documents processed via RAG

// 08

Integrating Prompt Engineering into CI/CD

8.1 The AI-augmented pipeline

Stage	LLM assistance	Prompt pattern
PR Created	Analyse diff, suggest relevant tests	"Given this diff [DIFF], which tests are most likely to catch regressions?"
Build	Generate missing unit tests for new functions	"Generate unit tests for these new functions: [FUNCTIONS]"
Test Exec	Classify failures: bug vs test issue vs flaky	"Classify this failure log [LOG]: product bug, test code issue, or flakiness?"
Code Review	Comment on testability of new code	"Identify code patterns reducing testability and suggest refactors"
Release Gate	Synthesise results into go/no-go	"Given these results [RESULTS], assess release risk as HIGH/MEDIUM/LOW"

8.2 Failure triage prompts

```
You are a QA engineer triaging a CI test failure.

Failing test: [TEST NAME]
Error message: [ERROR]
Stack trace:   [STACK TRACE]
Last 3 results: [PASS, PASS, FAIL]
Recent commits: [COMMIT MESSAGES]

Classify as exactly one of:
- PRODUCT_BUG: Application code is broken
- TEST_BUG: Test code is incorrect or outdated
- ENVIRONMENT: Infrastructure, network, or data issue
- FLAKY: Non-deterministic based on history

Output JSON only: { classification, confidence, rationale, nextAction }
```

// 09

Prompt Governance and Token Cost Management

9.1 The prompt library pattern

A prompt is code. It belongs in source control with a changelog, reviews, and regression tests. Structure it as a typed TypeScript module consumed by all teams:

```
// packages/prompt-library/src/test-generation.ts
export const prompts = {
  testGeneration: {
    fromAC: (ac: string, pom: string) => `
      You are a senior SDET. Given:
      Acceptance Criteria: ${ac}
      Page Object: ${pom}
      Generate Playwright TypeScript tests...`,
    failureTriage: (log: string, history: string) => `...`,
    apiFromSpec: (spec: string) => `...`,
  },
} as const;
```

9.2 Token budget controls that actually work

- Context compression: extract only relevant sections of large specs before injection
- Response caching: cache LLM responses for identical inputs. 40-60% hit rates are achievable in stable codebases
- Model tiering: use smaller, cheaper models for high-volume structured tasks. Reserve larger models for complex reasoning
- Batch processing: aggregate test generation requests. Azure Batch API reduces cost by 50%

Metric	Target
Cost per test generated	< $0.05 per test case
Cost per CI build (AI stages)	< $0.50 per build
Cache hit rate	> 40% for stable specs
Token budget per PR	< 50,000 tokens across all AI stages
Monthly AI spend vs. QA capacity	> 10x ROI (hours saved vs. cost)

// 10

Advanced Prompt Engineering Techniques

10.1 RAG-enhanced test generation

Retrieval-Augmented Generation lets your prompts access a knowledge base of existing tests, past defects, and institutional knowledge at query time, without stuffing the whole knowledge base into every prompt.

```
const relevantTests = await vectorDB.query({
  embedding: await embed(userStory),
  topK: 5,
  filter: { feature: featureName, type: 'regression' }
});

const prompt = `
  Generate new test cases for: ${userStory}
  CONTEXT -- Similar existing tests for reference:
  ${relevantTests.map(t => t.content).join('\n---\n')}
  DO NOT duplicate these existing tests.
  DO fill the coverage gaps these tests leave.
`;
```

10.2 Self-correcting prompts

```
Generate a Playwright test suite for: [FEATURE]

After generating, review your output against these criteria:
[ ] All tests use async/await correctly
[ ] No hardcoded test data (use fixtures or factories)
[ ] Each test has exactly one logical assertion group
[ ] All selectors follow ARIA-first strategy
[ ] No test depends on another test's side effects

If any criterion fails, revise before outputting.
Output the final corrected test suite only.
```

10.3 Prompt chaining for complex scenarios

01. Dependency Mapping: 'Identify all system dependencies and the order they must be exercised'
02. State Design: 'Given these dependencies [OUTPUT], design the test data state required at each step'
03. Test Code Generation: 'Generate Playwright code for this scenario [CHAIN 1 + CHAIN 2 OUTPUT]'
04. Review: 'Review this test code against our Test Quality Constitution'

05. Documentation: 'Generate Xray BDD test case documentation from [CHAIN 3 OUTPUT]'

// 11

Anti-Patterns and Troubleshooting

11.1 The 10 most dangerous QE prompting anti-patterns

01. The vague task

Problem: "Write tests for our login page." The model has no context about your framework, selector strategy, or output format.

Fix: Always specify framework, language, pattern, selector strategy, and provide a concrete example of the expected output structure.

02. Hallucination acceptance

Problem: Trusting AI-generated tests without running them. LLMs confidently generate calls to methods that don't exist.

Fix: Non-negotiable rule: every generated test must execute before committing. Treat unexecuted AI code as untested code.

03. Context overload

Problem: Injecting your entire codebase, full OpenAPI spec, and all existing tests into one prompt.

Fix: Extract only the minimum necessary context. Testing one endpoint? Inject only that endpoint's definition.

04. One-shot for complex scenarios

Problem: Asking for a 20-scenario E2E suite in a single prompt. Output quality degrades significantly.

Fix: Generate 5 scenarios at a time maximum. Use prompt chaining for complex multi-system scenarios.

05. Ignoring model drift

Problem: Assuming a prompt that worked with one model version produces equivalent results after an update.

Fix: Maintain a prompt regression test suite. Run it against any model version change before updating pipelines.

06. No human at high-risk gates

Problem: Fully automating generation and deployment for HIGH-risk, regulatory, or security-related coverage.

Fix: Mandatory human review for HIGH and CRITICAL risk scenarios regardless of AI confidence level.

07. Self-healing as communication substitute

Problem: Using self-healing automation instead of fixing unstable selectors or talking to developers.

Fix: Every self-heal event must trigger a communication to the owning team and a ticket to stabilise the selector.

08. Sharing sensitive data in prompts

Problem: Including production user data, real API keys, or PII in prompt context when using external LLM APIs.

Fix: Anonymise or synthesise all data before injection. Consider on-premise LLM deployment for sensitive contexts.

09. The magic wand expectation

Problem: Expecting prompt engineering to replace experienced QA judgment.

Fix: Position prompt engineering as a force multiplier for experienced engineers, not a replacement for QA expertise.

10. No performance measurement

Problem: Using prompts in production without measuring output quality over time.

Fix: Track compilation rate, execution rate, and defect detection rate. Monitor trends and retrain when metrics decline.

11.2 Troubleshooting guide

Symptom	Diagnosis and fix
Tests compile but fail immediately	Model used incorrect API or selector. Add explicit API version to context. Provide actual DOM snapshot.
Tests too generic	Insufficient few-shot examples. Add 2-3 concrete examples of your team's test style.
Output not reproducible	Prompt has ambiguity. Increase specificity. Set temperature=0 for deterministic output.
Only happy path coverage	Task instruction too broad. Explicitly: "Generate 5 happy path AND 10 negative tests".
Costs escalating in CI/CD	No token budget controls. Implement context compression, response caching, and model tiering.
Inconsistent quality across teams	No shared prompt library. Centralise prompts as typed TypeScript with versioning.

// 12

Building a Prompt Engineering Practice

12.1 The 90-day adoption roadmap

Days 1-30: Foundation

- Identify 3 high-value, low-risk use cases for initial adoption (API test generation from existing specs is a great start)
- Select 2-3 pilot engineers with strong testing fundamentals and genuine interest in AI tooling
- Establish a shared prompt library repository with version control
- Run a prompt engineering workshop covering zero-shot, few-shot, and chain-of-thought for your tech stack
- Create your first 3 team-standard prompt templates for the most common test generation tasks

Days 31-60: Integration

- Integrate failure triage prompting into CI/CD for pilot teams
- Establish token cost monitoring dashboard
- Run prompt regression tests for your first 10 production prompts
- Conduct retrospective: measure compilation rate, execution rate, coverage improvement

Days 61-90: Scale

- Roll out to all teams with training and shared prompt library access
- Implement approval gates for AI-generated tests in CRITICAL risk areas
- Launch an internal prompt engineering community of practice (bi-weekly 30-minute session)
- Set 6-month targets: test generation time reduction, coverage improvement, defect escape rate

12.2 Measuring the impact

KPI	Baseline	Target
Test authoring time	Hours per feature	50% reduction within 6 months
Test coverage	Current coverage reports	10pp improvement on new features

KPI	Baseline	Target
Defect escape rate	Defects found post-release	20% reduction
Time to triage CI failure	30 min (manual)	5 min (AI-assisted)
Prompt quality score	% of generated tests that compile + run	90%+ within 90 days
AI testing cost per sprint	Token costs per sprint	< $150 per team per sprint

// 13

Claude Code for Quality Engineering

13.1 What Claude Code changes for QA teams

Claude Code is Anthropic's agentic coding assistant that operates directly in your terminal, reads and writes files in your repository, executes commands, and runs tests in response to natural language instructions. For QE teams, this is a qualitative shift from chat-based prompt engineering to agentic automation.

Capability	QE application
File read/write	Creates and updates Page Objects, test files, fixtures, and configuration directly in your repo
Command execution	Runs Playwright, Jest, k6 and reads actual test output to inform next steps
Repository awareness	Understands your existing test architecture before generating (no hallucinated imports)
CLAUDE.md instructions	Reads team coding standards from a file in your repo root, persistent across all sessions
MCP server integration	Connects to Jira, GitHub, Datadog, and other systems as tools during task execution

13.2 CLAUDE.md: your persistent team prompt

CLAUDE.md is a file at the root of your repository that Claude Code reads at the start of every session. Time spent writing it well compounds across every engineer and every session.

```
# CLAUDE.md -- QA Team Standards

## Repository Context
Framework: Playwright 1.42+ | Language: TypeScript 5.x | Node: 20+
CI: Azure DevOps | Auth: ForgeRock OAuth2

## Coding Standards (MANDATORY)
- NEVER use CSS selectors, XPath, or nth-child locators
- ALWAYS use getByRole() as primary; getByLabel() second
- NEVER hardcode test data -- use DataFactory.create*()
- Tests MUST be independent: no shared browser state

## When Generating Tests
1. Read the relevant Page Object in /tests/pages/ first
2. Run: npx playwright test [file] --reporter=line to validate
3. Fix ALL failures before reporting done
```

```
## Approval Gates
- Tests tagged @critical: require human review before commit
- Any DELETE or data-modifying test: STOP and ask
```

// 14

Building QA Agents with Claude API

14.1 Agent architecture fundamentals

A Claude-powered QA agent uses the Anthropic API in a loop: call Claude with a task and available tools, Claude either calls a tool or produces a final answer, the program executes the tool call and returns the result, Claude continues reasoning. This continues until Claude signals task completion.

14.2 The Xray BDD Test Generator Agent

```
import Anthropic from '@anthropic-ai/sdk';

const client = new Anthropic();

async function runXrayAgent(storyKey: string) {
  const messages = [{
    role: 'user',
    content: `For story ${storyKey}:
      1. Fetch the story and acceptance criteria
      2. Check existing Xray tests to avoid duplication
      3. Generate Gherkin: happy path + negative + boundary
      4. Create each scenario linked to the story
      5. Report: tests created, coverage, untestable ACs`
  }];

  while (true) {
    const response = await client.messages.create({
      model: 'claude-opus-4-5', max_tokens: 4096, tools, messages
    });
    if (response.stop_reason === 'end_turn') break;
    // handle tool calls and append results...
  }
}
```

14.3 The Playwright Healer Agent

The Healer Agent monitors CI for selector-related failures and proposes minimal-diff fixes. Strict scope: it may only change locators, never assertions or test logic.

```
const HEALER_SYSTEM_PROMPT = `
You are a Playwright test healer. Your ONLY job: fix broken selectors.

RULES (non-negotiable):
1. Only modify: getByRole, getByLabel, getByText, locator()
2. Never change: assertions, test logic, data, imports, test names
3. Read the current DOM via the snapshot tool before fixing
4. Prefer ARIA-based locators over data-testid
```

```
5. Output a unified diff. DO NOT apply it. Human reviews first.
6. If the correct selector is unclear: output CANNOT_HEAL + reason
`;
```

// 15

Production-Grade Playwright Examples

These examples reflect real patterns from enterprise QE teams. Every example follows CLAUDE.md standards: ARIA-first selectors, fixture-based test isolation, API-driven setup/teardown, and typed Page Objects.

15.1 Full Page Object: Search Page

```
// tests/pages/SearchPage.ts
export class SearchPage {
  readonly searchInput: Locator;
  readonly resultsGrid: Locator;
  readonly loadingSpinner: Locator;

  constructor(private readonly page: Page) {
    this.searchInput = page.getByRole('searchbox', { name: 'Search products' });
    this.resultsGrid = page.getByRole('region', { name: 'Search results' });
    this.loadingSpinner = page.getByRole('progressbar');
  }

  async search(filters?: SearchFilters): Promise<void> {
    if (filters) await this.applyFilters(filters);
    await this.searchButton.click();
    await expect(this.loadingSpinner).toBeHidden({ timeout: 10_000 });
  }
}
```

15.2 Typed Playwright Fixtures

```
// tests/fixtures/index.ts
export const test = base.extend<TestFixtures>({
  tokenCache: [async ({}, use) => {
    const cache = new TokenCache();
    await use(cache);
    cache.clear();
  }, { scope: 'worker' }],

  authenticatedPage: async ({ page, tokenCache }, use) => {
    const token = await tokenCache.getOrFetch('test-user-1');
    await page.setExtraHTTPHeaders({ Authorization: `Bearer ${token}` });
    await use(page);
  },

  testItem: async ({ request }, use) => {
    const item = await ItemFactory.create(request, token);
    await use({ id: item.id });
    await ItemFactory.delete(request, token, item.id); // always clean up
```

```
  },
});
```

15.3 Enterprise playwright.config.ts

```
export default defineConfig({
  testDir: './tests/specs',
  fullyParallel: true,
  retries: process.env.CI ? 2 : 0,
  workers: process.env.CI ? 4 : undefined,
  reporter: [
    ['html', { outputFolder: 'playwright-report' }],
    ['junit', { outputFile: 'test-results/results.xml' }],
    ['playwright-spec-doc-reporter', { aiAnalysis: true }]
  ],
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'mobile-chrome', use: { ...devices['Pixel 7'] } },
  ],
  globalSetup: './tests/global-setup.ts',
});
```

// 16

The Complete AI-Augmented QE Workflow

16.1 End-to-end: from story to production

Stage	Who / What	Action
1. Story grooming	Claude Code + CLAUDE.md	Reads story, flags untestable ACs, asks PO for clarification before proceeding
2. Analyst Agent	Xray BDD Generator	Fetches story from Jira, checks existing tests, generates Gherkin: happy + negative + boundary
3. Approval Gate	QA Lead (human)	Reviews Analyst JSON. Approves or comments. Nothing proceeds without sign-off.
4. Writer Agent	Claude Code subagent	Reads approved plan and existing POMs, generates Playwright spec, runs tests, fixes failures
5. PR Review	Claude Code + GitHub MCP	Auto-generates review comments on testability, selector quality, missing negatives
6. CI Execution	Azure DevOps + Sentinel	PRODUCT_BUG = auto Jira ticket. TEST_BUG = Healer proposes diff.
7. Healer Gate	QA Lead (human)	Reviews Healer diff. Approves or rejects. Diffs are never auto-applied.
8. Release Gate	Oracle Agent	Synthesises sprint results and DORA metrics into go/no-go report with risk assessment

16.2 The twelve golden rules

01. Run everything. No AI-generated code goes to production without being executed.
02. Version your prompts. A prompt is code. It belongs in source control with a changelog.
03. Measure output quality. Track compilation rate, execution rate, and defect detection rate per prompt.
04. Keep humans at gates. AI generates. Humans approve for anything CRITICAL, data-destructive, or regulatory.
05. Never inject PII. Anonymise all data before it enters any external LLM API call.
06. Tier your models. Save expensive models for complex reasoning. Cheaper models for high-volume structured tasks.
07. Cache aggressively. Identical prompt and context must never call the API twice in the same sprint.

08. Scope your Healer. Self-healing agents touch only locators. Assertions and logic are human territory.
09. Govern your agents. Every agent needs a written constitution it cannot violate.
10. Monitor drift. LLM output quality changes with model updates. Run prompt regression before upgrading.
11. Own the cost. AI token spend is an engineering cost like cloud compute. Track it, budget it, optimise it.
12. Invest in CLAUDE.md. Time spent writing team standards there compounds across every engineer and session.

// A

Quick Reference Prompt Templates

A.1 Acceptance Criteria to Test Cases

```
ROLE: Senior SDET specialising in [DOMAIN] testing.

Given story: [USER STORY]
Given ACs:   [ACCEPTANCE CRITERIA]
Given POM:   [PAGE OBJECT CLASS]

Generate Playwright TypeScript tests:
- Cover all ACs (tag each test with AC ID)
- AAA pattern with comments
- ARIA-first selectors
- Independent (no shared state)
- 1 happy path, 2 negative, 1 boundary per AC

Flag any untestable ACs.
```

A.2 API Test from OpenAPI

```
ROLE: API test automation specialist.

Endpoint: [METHOD] [PATH]
Spec:     [ENDPOINT SPEC ONLY]
Auth:     [AUTH TYPE]

Generate Supertest/Jest tests:
- All documented response codes
- Schema validation for 200 response
- Auth: valid, expired, missing token
- Inputs: required missing, type violations
- Use: import { getAuthToken } from '../helpers/auth'

Output compilable TypeScript only. No explanations.
```

A.3 Failure Triage

```
ROLE: QA engineer triaging CI failure.

Test:    [TEST NAME]
Error:   [ERROR MESSAGE]
Stack:   [STACK TRACE]
History: [LAST 5 RESULTS]
Commits: [RECENT COMMITS]

Classify: PRODUCT_BUG | TEST_BUG | ENVIRONMENT | FLAKY
```

```
Output JSON: { classification, confidence, rationale, nextAction }
```

A.4 Model settings by task

Use case	Settings
Test generation (code)	Temperature: 0.1-0.3. Low temperature for consistent, compilable output.
Test strategy / brainstorm	Temperature: 0.7-0.9. Higher temperature for creative coverage exploration.
Failure classification	Temperature: 0. Deterministic classification requires zero temperature.
Security test generation	Temperature: 0.5. Balance creativity with specificity.
Executive reporting	Temperature: 0.4. Factually grounded with some linguistic variation.

A.5 Prompt engineering checklist

- Role defined with specific domain expertise
- Sufficient context provided (spec, story, existing code)
- Task instruction uses must/must not, not should/could
- Output format specified with a concrete example
- At least one few-shot example for code generation tasks
- Constraints explicitly listed (selectors, frameworks, conventions)
- Self-correction criteria included for complex tasks
- Sensitive data anonymised before injection
- Token budget appropriate for the model tier selected
- Output will be validated (compiled and executed) before use

Prompt engineering is not a silver bullet, and it's not a passing trend. It's an emerging engineering discipline that rewards the same qualities that make great QA engineers: rigour, systematic thinking, attention to detail, and the intellectual honesty to test your own assumptions.

The organisations that build genuine competitive advantage from AI in quality engineering will not be those that throw prompts at a chat interface and hope for the best. They'll be the ones that treat prompt engineering as a first-class engineering practice, with version control, testing, governance, and continuous improvement built in from the start.

The techniques in this guide are not theoretical. They are in production across enterprise engineering organisations today, delivering measurable improvements in test coverage, delivery velocity, and defect escape rates. Apply them systematically, measure the results honestly, and iterate.

```
// closing thoughts
$ cat wisdom.txt

"Quality Engineering has always been the discipline
that asks the hard questions before the user has to.
With prompt engineering in your toolkit, the scope of
those questions has expanded dramatically."

-- Pankaj Nakhat
```

pankajnakhat.com | linkedin.com/in/pankajnakhat | Abu Dhabi, UAE