Exam : **CCA-F**

Title : Claude Certified Architect Foundations (CCA-F)

Version : DEMO

1.When designing an automated code review system for a large repository, you notice that developers are experiencing 'alert fatigue' because the AI flags too many non-issues and stylistic preferences.
What is the most effective prompt engineering technique to reduce these false positives?
A. Instruct the model to be thorough and flag everything suspicious to ensure no bugs are missed.
B. Define explicit, measurable review criteria, such as flag functions exceeding 50 lines of code, instead of relying on vague instructions.
C. Set the model's temperature to 0.0 to ensure deterministic output and eliminate all false positives.
D. Provide at least 15 few-shot examples covering every possible coding style in the repository.
Answer: B
Explanation:
Explicit, measurable criteria produce consistent, actionable results that build developer trust and reduce false positives. Vague instructions like 'be thorough' lead to over-flagging and alert fatigue. Temperature adjustments do not fix vague criteria, and providing 15 few-shot examples bloats the prompt without proportional benefit.

2.You are building a classifier to categorize customer reviews, but the model struggles with ambiguous scenarios like sarcasm and mixed sentiments. You decide to use few-shot prompting.
What is the recommended best practice for this scenario?
A. Provide exactly 1 example of a standard positive review to establish the JSON format.
B. Provide 8 to 10 examples to ensure every possible sentiment category is covered comprehensively.
C. Provide 2 to 4 examples, ensuring that at least one example specifically addresses an ambiguous edge case like sarcasm.
D. Avoid few-shot prompting and instead use a detailed system prompt explaining the exact definition of sarcasm.
Answer: C
Explanation:
The optimal number of few-shot examples for ambiguous tasks is 2-4. Providing more than 6 examples bloats the prompt without adding proportional value. It is critical that at least one of these examples covers an edge case or ambiguous scenario to establish the expected reasoning pattern.

3.A pull request modifies 14 files across a stock tracking module. When analyzing all files in a single pass, Claude provides superficial comments, misses obvious bugs, and produces contradictory feedback.
How should you redesign the multi-pass review architecture to resolve this?
A. Switch to a higher-tier model with a larger context window so all 14 files receive adequate attention in one pass.
B. Require developers to manually split large PRs into smaller submissions of 3-4 files before triggering the automated review.
C. Run three independent single-pass reviews on the full PR and only flag issues that achieve consensus across at least two runs.
D. Split the review into focused passes: analyze each file individually for local issues, followed by a separate integration-focused pass for cross-file data flow.
Answer: D
Explanation:

Splitting reviews into focused passes directly addresses attention dilution when processing many files at once. Per-file analysis ensures consistent depth for local issues, while a separate integration pass catches cross-file architectural issues.

4.In a Continuous Integration (CI) pipeline, you run a script where a single Claude session generates a block of code and then immediately reviews its own changes in the same session.
Why is this considered an architectural anti-pattern?
A. The reviewer retains the reasoning context from the code generation phase, creating confirmation bias and reducing the likelihood of catching subtle issues.
B. It exceeds the maximum token limit for the context window, causing the session to silently truncate the generated code.
C. Same-session reviews trigger rate limiting automatically, breaking the CI/CD pipeline.
D. The Message Batches API does not support multi-turn tool calling within a single request.
Answer: A
Explanation:
Same-session self-review is an anti-pattern because the model retains its original reasoning context, creating a blind spot. Independent review instances without prior reasoning context are much more effective at objectively evaluating code.

5.While configuring prompt engineering standards for a production application, you must decide when to implement few-shot examples.
Which of the following scenarios represent the most effective use cases for few-shot prompting? Choose 2 correct answers.
A. Ambiguous classification tasks where the boundaries between categories are fuzzy or subjective.
B. Simple, well-defined data extraction tasks with clear objective criteria.
C. Tasks requiring standard output formats like valid JSON, where tool_use is already implemented.
D. Tasks requiring a custom, non-standard output format that Claude does not handle natively.
Answer: A, D
Explanation:
Few-shot prompting is most valuable when the task has ambiguous boundaries (like detecting sarcasm) or requires custom output formats that aren't standard. It is generally unnecessary for simple, well-defined tasks or standard formats like JSON which are better handled natively or via tool_use.