Domain 3 — 20%

Claude Code Configuration & Workflows

Domain 3 — 20% of the CCA exam

IN THIS MODULE:

- CLAUDE.md hierarchy: project, directory, user levels
- Slash commands and skills with frontmatter
- context: fork for isolated sub-agents
- CI/CD integration with -p flag and --output-format json
- Iterative refinement techniques

90 PRACTICE QUESTIONS — 30 BEGINNER · 30 INTERMEDIATE · 30 ADVANCED

QUESTION BANK OVERVIEW

Level	Questions	Focus
Beginner	Q1–Q30	Core concepts, terminology, and foundational patterns
Intermediate	Q31–Q60	Application scenarios, design decisions, trade-offs
Advanced	Q61–Q90	Production architecture, edge cases, system design

HOW TO USE THIS MODULE

Work through each level in order. For Beginner questions, aim for 90%+ before moving to Intermediate. For Intermediate, 80%+ before Advanced. Each question includes the correct answer and a full explanation — read the explanation even for questions you answered correctly to understand the underlying principle. The exam tests judgment, not memorization.

BEGINNER QUESTIONS

Questions 1–30

Q1. What is the purpose of CLAUDE.md files in Claude Code?

A) They store Claude Code's configuration settings like API keys and model versions

B) They provide project context, conventions, and instructions that Claude Code loads automatically to guide its behavior on that codebase

C) They are the output files Claude Code generates after analyzing a project

D) They define the tools Claude Code can use in a project

■ Correct Answer: B

CLAUDE.md files are persistent context for Claude Code — they tell it about the project architecture, coding standards, testing requirements, and team conventions. Loading automatically means Claude Code always knows the project context without you having to re-explain it each session.

Q2. Which CLAUDE.md location is shared with the entire team via version control?

A) ~/.claude/CLAUDE.md

B) /tmp/CLAUDE.md

C) .claude/CLAUDE.md (or root CLAUDE.md) in the repository

D) ~/Documents/CLAUDE.md

■ Correct Answer: C

Project-level CLAUDE.md lives in the repo and is committed to git. Every team member who clones the repo gets the same Claude Code configuration. User-level (~/.claude/CLAUDE.md) is personal and never shared.

Q3. What does the @import syntax in CLAUDE.md do?

A) It imports Python modules for Claude Code to use

B) It references external files so CLAUDE.md stays modular — rules files are maintained separately and included at load time

C) It imports configuration from another project's CLAUDE.md

D) It loads MCP server definitions

■ Correct Answer: B

@import .claude/rules/testing.md lets you maintain testing standards separately from API conventions. Each concern lives in its own file, making the configuration maintainable and allowing different teams to own different rule files.

Q4. What is the .claude/rules/ directory used for?

A) Storing Claude Code execution logs

B) Organizing topic-specific rule files that can be selectively loaded based on file path patterns

C) Caching Claude Code responses for faster repeated queries

D) Storing tool definitions for Claude Code

■ Correct Answer: B

.claude/rules/ provides organized topic-specific configuration: testing.md, api-conventions.md, deployment.md, security.md — each focused on one concern. Files in this directory can use YAML frontmatter to scope loading to specific file path patterns.

Q5. What flag do you add to run Claude Code non-interactively in a CI pipeline?

A) --ci

B) --no-input

C) -p (or --print)

D) --batch

■ Correct Answer: C

The -p (--print) flag runs Claude Code in print mode: it processes the prompt, outputs the result, and exits. Without -p, Claude Code waits for interactive input — which causes CI pipelines to hang indefinitely.

Q6. When is plan mode appropriate vs. direct execution?

A) Always use plan mode — it produces better results for all tasks

B) Plan mode for complex, multi-file, architectural tasks. Direct execution for well-scoped, single-file, clear-scope changes

C) Direct execution is always faster; plan mode is for documentation only

D) Plan mode requires senior developer approval; direct execution is self-service

■ Correct Answer: B

Plan mode adds a design phase before execution: Claude maps the full scope, considers approaches, presents a plan. This is valuable for refactors affecting 45+ files or architectural decisions. For single-file bug fixes, plan mode adds overhead without benefit.

Q7. What does context: fork do in a skill's frontmatter?

A) Creates a backup copy of the current session before running the skill

B) Runs the skill in an isolated sub-agent context, preventing verbose skill output from polluting the main session's context window

C) Forks the Git repository before making any changes

D) Enables parallel execution of multiple skills simultaneously

■ Correct Answer: B

Skills with context: fork run in a isolated sub-context. All their verbose output (thousands of lines of file analysis, brainstorming, etc.) stays in the fork. The main session receives only the distilled summary, keeping your main context clean and focused.

Q8. What is the difference between a slash command and a skill in Claude Code?

A) Slash commands are for developers; skills are for non-technical users

B) Skills have frontmatter configuration (context, allowed-tools, argument-hint) and can run in isolated contexts; slash commands are simpler prompt templates without these capabilities

C) Slash commands execute code; skills only generate text

D) There is no difference — they are the same feature with different names

■ Correct Answer: B

Slash commands are lightweight prompt templates invoked via /command-name. Skills are more powerful: frontmatter enables context isolation (fork), tool restrictions (allowed-tools), and argument prompting (argument-hint). Use skills for complex, configurable workflows.

Q9. Where should project-wide slash commands be stored to share them with the team?

A) ~/.claude/commands/ on each developer's machine

B) .claude/commands/ in the project repository (committed to version control)

C) In the root CLAUDE.md file as command definitions

D) In a .commands file in the project root

■ Correct Answer: B

.claude/commands/ in the repo is version-controlled → shared with the team. Personal commands go in ~/.claude/commands/ and are never shared. Team commands like /review-pr or /run-tests belong in the project scope.

Q10. What does the allowed-tools frontmatter in a skill do?

A) Lists which team members can use this skill

B) Restricts which tools the skill can invoke during its execution, limiting scope and preventing destructive actions

C) Specifies which MCP servers the skill connects to

D) Defines which file types the skill can read

■ Correct Answer: B

allowed-tools constrains the skill's capabilities: an analysis skill restricted to Read, Grep, Glob cannot accidentally modify files. A reporting skill restricted to Write cannot browse the web. This is the principle of least privilege applied to skills.

Q11. What does the /memory command do in Claude Code?

A) Clears Claude Code's memory to start fresh

B) Shows which CLAUDE.md and memory files are currently loaded, helping diagnose inconsistent behavior

C) Exports the current session context to a file

D) Imports a previous session's context into the current one

■ Correct Answer: B

/memory reveals what Claude Code is currently working with: which CLAUDE.md files are loaded and their content. This is essential for debugging — if Claude Code is ignoring conventions, /memory shows whether the relevant file is loaded.

Q12. What is the argument-hint frontmatter field in a skill used for?

A) Documenting the skill for the team wiki

B) Displaying a prompt to the developer when they invoke the skill without arguments, asking them to provide required parameters

C) Setting default argument values

D) Restricting which argument types are valid

■ Correct Answer: B

argument-hint: 'Enter the module name to analyze' appears as a prompt when a developer invokes the skill without arguments. This prevents skills from failing silently when required parameters are missing.

Q13. What is the recommended approach when a task spans multiple directories with different conventions (e.g., frontend React code and backend Python code)?

A) Put all conventions in the root CLAUDE.md

B) Create directory-level CLAUDE.md files in each relevant subdirectory, or use path-scoped rules in .claude/rules/ for file-type-specific conventions

C) Create separate Claude Code projects for frontend and backend

D) Use a single shared convention file and ignore the differences

■ Correct Answer: B

Directory-level CLAUDE.md files (frontend/CLAUDE.md, backend/CLAUDE.md) provide subsystem-specific context. Alternatively, path-scoped rules in .claude/rules/ with glob patterns (paths: ['frontend/**/*.tsx']) scope conventions to specific files without requiring directory structure changes.

Q14. What is the purpose of --output-format json in Claude Code CI integration?

A) Compresses the response to reduce file size

B) Produces machine-parseable structured output that can be automatically processed to post inline PR comments or integrate with review tools

C) Formats the output for display in the GitHub Actions log

D) Enables JSON input for the review criteria

■ Correct Answer: B

--output-format json combined with --json-schema produces structured findings that can be parsed programmatically. Your CI script can then post each finding as an inline comment on the specific PR diff line — a much better UX than a blob of text.

Q15. Where do personal Claude Code customizations live that you DON'T want to share with teammates?

A) .claude/personal/CLAUDE.md in the project

B) ~/.claude/CLAUDE.md (user-level, never committed to version control)

C) A .gitignore'd CLAUDE.local.md in the project root

D) Personal settings are not supported — all settings are team-wide

■ Correct Answer: B

~/.claude/CLAUDE.md is your personal configuration space: personal aliases, formatting preferences, development shortcuts. It never enters the repo. Project-level settings that should apply to everyone go in .claude/CLAUDE.md.

Q16. Why is using an independent Claude Code session to review a PR more effective than self-review?

A) Independent sessions use a more powerful model

B) The session that wrote the code retains generation reasoning context, making it biased toward validating its own decisions; a fresh session has no such bias

C) Independent sessions have access to more tools

D) Self-review is not supported in Claude Code

■ Correct Answer: B

This is a core reliability principle: the session that generated the code 'remembers' its reasoning and is less likely to question its own decisions. An independent review session sees only the code, not the reasoning that produced it — it catches more issues.

Q17. What is the Explore subagent in Claude Code and when should you use it?

A) A subagent specialized in web browsing and external research

B) A subagent that runs verbose discovery phases in an isolated context to prevent main session context exhaustion during multi-phase tasks

C) The default subagent used for all Claude Code operations

D) A monitoring subagent that reports on Claude Code's resource usage

■ Correct Answer: B

Explore runs discovery (reading files, grepping code, understanding architecture) in a fork, accumulating potentially thousands of tokens of tool output. The main session gets only the distilled findings. This preserves main session context budget for the implementation phase.

Q18. What should a CLAUDE.md file include about a codebase?

A) Only the list of approved coding tools

B) Project overview and tech stack, architectural decisions and patterns, testing standards, deployment process, and what to avoid (anti-patterns, banned practices)

C) Employee names and contact information for escalation

D) Only the sections relevant to whatever task Claude Code is currently working on

■ Correct Answer: B

CLAUDE.md should give Claude Code the context a senior developer would know: what the system does, key architectural decisions, how to test, how to deploy, and common pitfalls. This is everything you'd tell a new team member on their first day.

Q19. What is the interview pattern for iterative refinement?

A) Claude Code interviews the developer about their qualifications

B) Having Claude ask clarifying questions to surface design considerations before implementing — especially useful in unfamiliar domains

C) Running a technical interview to assess code quality

D) Generating multiple solution variants and asking the developer to pick one

■ Correct Answer: B

The interview pattern: ask Claude to interview you before implementing. 'Before writing this caching layer, ask me the questions you need answered.' Claude will surface considerations you may not have thought of (invalidation strategy, failure modes, consistency guarantees) before you're committed to an implementation.

Q20. When should you provide test-driven feedback for iterative refinement?

A) Only after the implementation is complete

B) Write tests first describing expected behavior, run them against Claude's implementation, then share specific failures for Claude to fix — this produces better outcomes than prose feedback

C) Only for junior developers — senior developers use direct execution

D) Test-driven feedback is not supported in Claude Code

■ Correct Answer: B

Tests are precise specifications. 'This test fails: expected {result: 42}, got {result: 0}' is more actionable than 'the calculation seems wrong'. Writing tests first also forces you to specify expected behavior precisely, which improves implementation quality.

Q21. A developer reports that their custom slash command works in their terminal but not in the team's shared project. What is the most likely cause?

A) Slash commands don't work in shared projects

B) The command file is in ~/.claude/commands/ (personal) instead of .claude/commands/ (project) — it's not committed to the repository

C) The command syntax has an error that only manifests in certain environments

D) Shared projects require admin approval for custom commands

■ Correct Answer: B

Personal vs project scope: ~/.claude/commands/ exists only on that developer's machine. .claude/commands/ is in the repo and available to all team members. Moving the file to .claude/commands/ and committing it makes it available to everyone.

Q22. What is the primary advantage of using Claude Code for complex refactoring compared to just using the chat interface?

A) Claude Code has access to a more powerful model

B) Claude Code can read and edit actual files on your filesystem, run tests, and iteratively fix issues — operating as an engineering partner, not just a text generator

C) Claude Code has a larger context window

D) Claude Code is cheaper per token

■ Correct Answer: B

Claude Code's file system integration is the key differentiator: Read, Write, Edit, Grep, Glob, and Bash let it work with your actual codebase — not just discuss code in the abstract. It can make changes, run tests, see failures, and iterate.

Q23. How does Claude Code know what version control system your project uses?

A) You must specify it in CLAUDE.md

B) Claude Code detects the version control system by examining the repository structure (.git, .hg directories) and uses appropriate Bash commands accordingly

C) Claude Code only supports Git

D) Version control is configured in the .claude/settings.json file

■ Correct Answer: B

Claude Code uses environmental awareness — it explores the project structure to understand what tools and systems are in use. Finding .git triggers Git-specific workflows. Documenting the VCS in CLAUDE.md is good practice but not required for basic detection.

Q24. What does it mean when Claude Code says it will 'run tests' as part of a task?

A) It simulates test execution mentally without actually running code

B) It uses the Bash tool to execute actual test commands in your project (npm test, pytest, go test) and reads the real output to verify its changes work

C) It generates test scenarios but leaves execution to the developer

D) It only runs tests if you have a test runner configured in CLAUDE.md

■ Correct Answer: B

Claude Code actually executes code via the Bash tool. Real test execution means real failures are caught: 'TypeError: undefined is not a function' in test output leads Claude Code to examine and fix the actual issue, not a hypothetical one.

Q25. What happens when you provide feedback to Claude Code on its implementation?

A) Claude Code resets and starts from scratch

B) Claude Code treats your feedback as new context and iteratively refines the implementation, informed by all prior attempts and your specific feedback on what needs changing

C) Feedback is logged but doesn't affect the current session

D) Claude Code only accepts feedback in a specific feedback format

■ Correct Answer: B

Iterative refinement is core to the Claude Code workflow. Feedback like 'This works but the error messages are confusing' or 'The tests pass but performance is poor on large inputs' is incorporated into subsequent iterations. Each iteration builds on prior context.

Q26. What is the difference between /review-pr and /security-audit slash commands if both are defined in .claude/commands/?

A) They are identical — command names are just labels

B) Each command file defines its own prompt and scope; /review-pr might focus on correctness and test coverage while /security-audit focuses on vulnerabilities — entirely different workflows despite being invoked similarly

C) Only one custom command can be active at a time

D) /security-audit requires elevated permissions to run

■ Correct Answer: B

Each .md file in .claude/commands/ defines a complete independent workflow. The command name is just the invocation — the actual prompt, scope, and output format are entirely defined by the file content. You can have unlimited purpose-specific commands.

Q27. What is the best way to tell Claude Code about a project's testing philosophy (e.g., 'we prefer integration tests over unit tests')?

A) Mention it in every prompt you give to Claude Code

B) Document it in CLAUDE.md (project-level or in a .claude/rules/testing.md file) so it's automatically loaded as persistent context every session

C) Store it in a .testing-philosophy file

D) Claude Code learns this automatically from your test file structure

■ Correct Answer: B

CLAUDE.md is for persistent project context that applies to all sessions. Testing philosophy ('we favor integration tests; each test must have comments for arrange/act/assert sections') belongs in CLAUDE.md so every Claude Code session starts with this knowledge.

Q28. What does 'allowlisting specific Bash commands' in Claude Code mean?

A) Approving Bash commands before they run

B) Configuring Claude Code to only use a specific subset of Bash commands — restricting which commands it can execute to reduce risk

C) A list of Bash commands that require elevated permissions

D) Commands that Claude Code is allowed to show output from

■ Correct Answer: B

In sensitive environments, you can restrict Claude Code's Bash access to a safe subset (e.g., allow: npm test, git status, grep but not: rm, sudo, curl). This reduces the risk of accidental or malicious destructive commands.

Q29. What is the -p flag's purpose in Claude Code?

A) Enables plan mode for the session

B) 'Print mode' — runs Claude Code non-interactively, processes the prompt, prints the result to stdout, and exits. Required for CI/CD pipeline integration.

C) Passes a custom prompt file to Claude Code

D) Enables verbose debug output

■ Correct Answer: B

-p (--print) is the CI integration flag: no interactive prompts, no waiting for input. Combine with --output-format json for machine-parseable output. Without -p, Claude Code expects an interactive terminal and will hang in CI pipelines.

Q30. What does context: fork prevent in a skill's execution?

A) Prevents the skill from being invoked by multiple agents simultaneously

B) Prevents the skill's verbose intermediate work (file reads, analysis, brainstorming) from accumulating in the main session's context window

C) Prevents the skill from modifying files outside its allowed directory

D) Prevents the skill from calling external APIs

■ Correct Answer: B

Fork isolation contains the mess: a skill that reads 100 files generates thousands of tokens of file content in its context. With context: fork, all that stays in the fork. The main session gets only the skill's final summary output, keeping the main context clean for subsequent work.

INTERMEDIATE QUESTIONS

Questions 31–60

Q31. A junior developer joins the team and reports that Claude Code isn't following the TypeScript naming conventions everyone else follows. Other team members have no issues. What is the most likely cause?

A) Claude Code needs to be updated to the latest version

B) The naming conventions are defined in the developer's user-level ~/.claude/CLAUDE.md on another machine; they need to be moved to the project-level .claude/CLAUDE.md

C) The conventions are correct but Claude Code is ignoring them intentionally

D) The developer needs to run /reload to load the latest CLAUDE.md

■ Correct Answer: B

User-level CLAUDE.md doesn't travel with the repo. If someone set up conventions on their own machine in ~/.claude/CLAUDE.md and others rely on that, new team members won't have them. Team conventions belong in the project-level .claude/CLAUDE.md.

Q32. Your .claude/rules/terraform.md should apply whenever editing any Terraform file anywhere in the project (not just in a specific directory). How should the frontmatter be written?

A) paths: ['terraform/']

B) paths: ['**/*.tf', 'terraform/**/*', '**/*.tfvars']

C) paths: ['*.tf'] — this matches all .tf files

D) No frontmatter needed — .claude/rules/ files always apply globally

■ Correct Answer: B

Glob patterns in paths: must match all locations where Terraform files appear. **/*.tf matches .tf files at any depth. terraform/**/* matches files in a dedicated directory. **/*.tfvars catches variable files. paths: ['*.tf'] only matches the root directory.

Q33. You want a CLAUDE.md for testing conventions that applies to test files in src/__tests__/, components//*.test.tsx, and e2e/. What is the correct path pattern?**

A) paths: ['tests/**']

B) paths: ['**/*.test.tsx', '**/*.test.ts', '**/*.spec.ts', 'e2e/**/*', 'src/__tests__/**']

C) paths: ['*.test.ts', '*.test.tsx']

D) A separate CLAUDE.md in each test directory

■ Correct Answer: B

Comprehensive glob coverage ensures the rule loads regardless of where tests live. Multiple patterns in the paths array are OR'd — if any pattern matches the currently edited file, the rule loads. Using multiple patterns is better than trying to write one complex pattern.

Q34. A Claude Code skill produces 15,000 tokens of codebase analysis output. What frontmatter setting prevents this from polluting the main session?

A) max-output: 1000

B) context: fork

C) output-mode: summary

D) isolated: true

■ Correct Answer: B

context: fork runs the skill in an isolated sub-context. The 15,000-token analysis stays in the fork. The main session receives only the summary returned by the skill. Without fork, that analysis consumes main session context budget permanently.

Q35. Your team wants a /security-audit slash command available to all developers. A senior developer also wants a personal variant with stricter rules. How is this implemented without affecting the team version?

A) Edit the shared command directly with a senior_mode flag

B) The senior developer creates a personal variant in ~/.claude/commands/security-audit-strict.md with a different name to avoid overriding the shared command

C) The senior developer maintains their own branch with a modified command

D) Personal command variants are not supported — shared commands apply to everyone

■ Correct Answer: B

Personal variants in ~/.claude/commands/ with different names coexist with team commands in .claude/commands/. The senior developer uses /security-audit-strict for stricter analysis; everyone else uses /security-audit. No git conflict, no teammate impact.

Q36. When including prior code review findings in a new review run, what instruction should you give Claude Code to avoid duplicate PR comments?

A) 'Ignore all previous reviews — start fresh'

B) 'Here are the findings from the prior review: [findings]. Report ONLY issues that are new or still unaddressed after the latest commits'

C) 'Output only findings with severity: critical'

D) 'Do not repeat any finding that contains the same file name as a prior finding'

■ Correct Answer: B

Contextualizing with prior findings enables differential reviews: Claude Code compares the current code state against known issues and reports only what's new or unresolved. Without this context, every review run re-reports all historical findings as new.

Q37. How does Claude Code use CLAUDE.md in a CI/CD pipeline where it's invoked with -p?

A) CLAUDE.md is not loaded in non-interactive mode

B) CLAUDE.md loads automatically regardless of interactive mode — CI-invoked Claude Code uses the same project context as developer sessions

C) You must pass --config CLAUDE.md explicitly in CI commands

D) CLAUDE.md is only loaded when a developer explicitly runs /memory

■ Correct Answer: B

CLAUDE.md is project context, not an interactive feature. It loads automatically whenever Claude Code starts in the project directory — interactive or not. This is what makes CI reviews consistent with developer workflow reviews.

Q38. You are using the iterative refinement technique with concrete examples. When should you provide 2-3 input/output examples vs. detailed prose instructions?

A) Always use prose — examples are harder to maintain

B) Use examples when prose descriptions are being interpreted inconsistently; examples are more effective than additional instructions for communicating ambiguous transformations

C) Use examples only for beginners — experienced developers use prose

D) Examples and prose are equally effective — choose based on preference

■ Correct Answer: B

Examples communicate transformation intent more precisely than prose. 'Transform address to standard format' is ambiguous. An example ('123 Main St. Apt 4' → '123 Main Street, Apt 4') is unambiguous. When prose produces inconsistent results, examples consistently fix it.

Q39. What is the best approach for addressing multiple issues found in a code review when some issues interact with each other?

A) Always fix issues one at a time in separate iterations

B) Provide all interacting issues in a single detailed message so Claude Code can consider their relationships; fix independent issues sequentially

C) Always provide all issues at once regardless of independence

D) Only provide the most critical issue — Claude Code can find the others

■ Correct Answer: B

Interaction awareness: if issue A (missing null check) and issue B (wrong return type) are related, fixing A changes the context for fixing B. Providing both together lets Claude Code reason about their relationship. Independent issues are fine to address sequentially — each fix is isolated.

Q40. A developer wants to use Claude Code to explore a legacy codebase before making changes, but doesn't want the exploration to use up context for the implementation phase. What is the correct workflow?

A) Use direct execution — exploration isn't needed before implementation

B) Use plan mode for exploration (or invoke the Explore subagent via a skill with context: fork), then start the implementation in the main session with the exploration summary

C) Run two separate Claude Code instances simultaneously

D) Use a larger max_tokens budget for exploration sessions

■ Correct Answer: B

Plan mode or the Explore subagent (context: fork) isolates discovery. Exploration accumulates large tool outputs (file reads, grep results). Keeping this in a fork or plan mode context preserves the main session's token budget for the actual implementation.

Q41. What information should CLAUDE.md include about your test suite to improve CI test generation?

A) A list of all existing test file names

B) Testing standards (what makes a good test), valuable test criteria (edge cases that matter), available fixtures and test helpers, and what is already covered (to avoid duplicates)

C) The test framework version and configuration only

D) Links to testing documentation on external websites

■ Correct Answer: B

Test generation quality depends on context: what to test (standards), how to test (fixtures and helpers), and what's already tested (to avoid duplicates). Without this context, Claude Code generates generic tests that may duplicate existing coverage or miss critical edge cases.

Q42. When should you use a directory-level CLAUDE.md vs. path-scoped rules in .claude/rules/?

A) Directory-level CLAUDE.md is always better — it's simpler to manage

B) Use directory-level CLAUDE.md for conventions tightly coupled to a specific directory tree. Use path-scoped rules when conventions apply to files scattered across multiple directories

C) Path-scoped rules always replace directory-level CLAUDE.md — never use both

D) Directory-level is for large teams; path-scoped is for small teams

■ Correct Answer: B

Directory-level CLAUDE.md works well when all affected files are in one subtree (frontend/). Path-scoped rules work better for cross-cutting concerns (all *.test.ts files regardless of location, all *.tf files regardless of directory). Both can coexist.

Q43. How does Claude Code know what existing tests cover when generating new tests in CI?

A) It automatically scans and understands all existing test files

B) You must explicitly provide the relevant existing test files in the CI invocation context, instructing Claude Code not to generate scenarios already covered

C) Test generation always creates completely new tests with no awareness of existing coverage

D) Claude Code reads a coverage report file automatically

■ Correct Answer: B

Claude Code has no automatic awareness of existing tests unless you provide them. In CI: pass relevant test files in context with instruction 'do not suggest scenarios already covered in these tests'. This prevents duplicate test generation.

Q44. A team has CLAUDE.md files at three levels: root, src/, and src/payments/. Which rules apply when Claude Code is editing src/payments/refund.py?

A) Only src/payments/CLAUDE.md — the most specific wins

B) All three: root CLAUDE.md + src/CLAUDE.md + src/payments/CLAUDE.md — all apply simultaneously, with more specific rules supplementing (not replacing) general ones

C) Only root CLAUDE.md — directory-level files are supplementary only when explicitly imported

D) The developer must specify which CLAUDE.md applies with a command flag

■ Correct Answer: B

CLAUDE.md files are additive, not overriding. All three apply: root provides general project context, src/ provides backend-specific rules, payments/ provides payment-specific rules (PCI compliance, decimal handling). More specific files add to, not replace, general ones.

Q45. You need to prevent a CI review from commenting on the same issue twice across multiple PR updates. What is the architectural solution?

A) Use a lower temperature to make reviews more deterministic

B) Maintain a review state store (database or PR comment API) that tracks which findings have been reported; inject prior findings into each Claude Code invocation with instructions to report only new/unresolved issues

C) Delete all prior PR comments before each new review run

D) Use a unique session ID per PR to maintain review continuity

■ Correct Answer: B

Stateful review management: store posted findings with their PR+file+line key. On each new commit, inject stored findings into context with 'report only new or still-unaddressed issues'. This prevents the review from re-reporting issues that were already discussed or fixed.

Q46. Why should test generation use the same Claude Code session that wrote the code vs. an independent session?

A) It should NOT — independent review sessions catch more gaps than self-review

B) Same session is better for test generation — it understands implementation intent and can write tests for internal behavior

C) Same session is always faster

D) Independent sessions cannot read the implementation files

■ Correct Answer: A

This is a trick question. For REVIEW, independent sessions are better. For TEST GENERATION, same-session understanding of implementation intent is actually useful — the session knows the code's intended behavior and edge cases. The exam tests whether you know the difference.

Q47. How do you handle a Claude Code task that spans a large refactor (45+ files) without context window exhaustion?

A) Increase max_tokens to accommodate all 45 files

B) Use plan mode first to create a structured refactor plan, then execute file-by-file in focused implementation sessions, using checkpoints to track progress

C) Process all 45 files simultaneously in a single Claude Code invocation

D) Limit refactors to 10 files maximum

■ Correct Answer: B

Large refactors require phased execution: (1) plan mode generates a complete refactor plan without touching files, (2) implementation executes the plan in manageable chunks (5-10 files per session), (3) checkpoint summaries maintain continuity between sessions. This avoids context overflow while maintaining coherence.

Q48. What is the most effective CLAUDE.md content for reducing Claude Code's false positives on a codebase that uses unconventional patterns intentionally?

A) A long list of rules Claude Code must follow

B) Explicit documentation of intentional patterns with the reasoning: 'We use X pattern because Y — do not flag this as an anti-pattern'

C) A list of file patterns Claude Code should not analyze

D) Reduced max_tokens to limit how much Claude Code checks

■ Correct Answer: B

Documenting intentional unconventional patterns with explanations prevents Claude Code from flagging them as bugs. 'We use global state in config.ts intentionally for performance — do not suggest dependency injection for this module' stops repeated false positive suggestions.

Q49. How should CI test generation handle a PR that modifies a function that is already covered by 15 existing tests?

A) Regenerate all 15 tests to ensure they match the new implementation

B) Provide the 15 existing tests as context; instruct Claude Code to analyze what's already covered and generate ONLY tests for behavior introduced or changed by this PR

C) Skip test generation when existing coverage is sufficient

D) Delete the old tests and generate fresh ones

■ Correct Answer: B

Differential test generation: provide existing tests as context and instruct Claude Code to find coverage gaps introduced by the PR changes specifically. Regenerating all 15 tests risks destabilizing working tests and wastes effort.

Q50. You want a skill that creates a standardized PR description. The skill should ask for the PR title if not provided, and should only use Read and Glob tools. Write the correct frontmatter.

A) tools: [Read, Glob] / prompt: 'Enter PR title'

B) allowed-tools: [Read, Glob] / argument-hint: 'Enter the PR title (or press enter to generate from branch name)'

C) permissions: [Read, Glob] / input-required: 'PR title'

D) restricted-tools: [Write, Bash] / argument-hint: 'PR title'

■ Correct Answer: B

allowed-tools restricts tools; argument-hint prompts for missing input. The correct frontmatter keys are exactly 'allowed-tools' and 'argument-hint'. Using incorrect key names means the frontmatter is silently ignored.

Q51. When is it appropriate to use 'plan mode' in Claude Code before making changes?

A) For all tasks — plan mode always produces better results

B) For tasks with architectural implications, multiple file changes, or where the approach isn't immediately clear. Skip for well-scoped single-file changes where the implementation is straightforward.

C) Only for tasks that take more than 30 minutes

D) Plan mode is only for team leads — individual developers use direct execution

■ Correct Answer: B

Plan mode adds value proportional to task complexity and uncertainty. A bug fix in a known location: direct execution is faster and equally good. Adding a new authentication provider across 20 files with database schema changes: plan mode prevents costly mistakes by mapping the full scope before touching any file.

Q52. You need Claude Code to generate documentation that always follows your team's specific doc template. How do you ensure this without manual review?

A) Tell Claude Code verbally each time to use the template

B) Include the template structure in CLAUDE.md with clear section descriptions and required fields; create a /generate-docs slash command that references the template explicitly; include a sample well-formatted doc as a few-shot example

C) Post-process all Claude Code outputs through a formatter

D) Use a fixed temperature setting for deterministic output

■ Correct Answer: B

Template compliance requires explicit template specification in CLAUDE.md plus a dedicated command that references it. The few-shot example doc eliminates ambiguity about formatting. A dedicated /generate-docs command creates a focused context for documentation tasks vs. general coding tasks.

Q53. How should CLAUDE.md handle information that changes frequently (like API endpoint URLs or version numbers)?

A) Hardcode all current values — CLAUDE.md should be complete and self-contained

B) Reference external configuration files or environment variables for volatile values; CLAUDE.md should describe structure and patterns, not frequently-changing specific values

C) Update CLAUDE.md daily with current values via a CI pipeline

D) Omit volatile information — Claude Code will discover it from the codebase

■ Correct Answer: B

CLAUDE.md is documentation, not configuration. Hardcoding version numbers creates maintenance burden — every version bump requires a CLAUDE.md update. Instead: 'API version is in config/api-config.json' or 'base URL is in .env'. CLAUDE.md describes patterns; code files contain current values.

Q54. What is the correct interpretation of Claude Code running in 'non-interactive mode' with -p?

A) Claude Code runs with reduced capabilities

B) Claude Code processes the prompt, executes the task, outputs results, and exits — no interactive prompts, STDIN reading, or clarifying questions. Suitable for CI/CD pipelines.

C) Non-interactive mode disables all file write operations for safety

D) Only read-only operations are available in non-interactive mode

■ Correct Answer: B

-p (print mode) is designed for scripted, automated use: Claude Code receives the full instruction upfront, executes without asking for clarification, and exits when complete. If Claude Code would normally ask a clarifying question, it instead makes its best judgment and proceeds.

Q55. What does the --json-schema flag in Claude Code CI invocations do?

A) Validates that the Claude Code configuration is valid JSON

B) Specifies a schema file that Claude Code must follow when producing --output-format json output, ensuring structured machine-parseable findings

C) Converts the CLAUDE.md to JSON format for CI consumption

D) Enables JSON-formatted logs for the CI pipeline runner

■ Correct Answer: B

--json-schema + --output-format json is the combination for structured CI output: the schema defines required fields (file, line, severity, message, category), and Claude Code's JSON output conforms to it. Your CI script parses the JSON and posts inline PR comments without text parsing.

Q56. You are writing a CLAUDE.md for a codebase that uses a monorepo pattern where shared utilities live in /packages/shared and are imported by both frontend and backend. What should CLAUDE.md emphasize?

A) Shared utilities should be duplicated in both frontend and backend to avoid dependencies

B) When modifying /packages/shared, always check for and update all imports in both frontend and backend; never make breaking changes to shared interfaces without a migration plan

C) Only senior developers should modify shared utilities

D) CLAUDE.md should list all current shared utility function signatures

■ Correct Answer: B

Shared package rules prevent breaking changes: Claude Code must understand that changes to /packages/shared have downstream effects across all consumers. This rule prevents localized fixes in shared that break consumers, and establishes the pattern for interface stability.

Q57. How does Claude Code's iterative refinement cycle work when a test fails after implementation?

A) Start over with a different implementation approach

B) Share the specific test failure (error message, expected vs actual, failing test code) as feedback; Claude Code uses this concrete failure information to diagnose and fix the specific issue

C) Reduce the requirements and implement a simpler version that passes

D) Mark the test as pending and skip it

■ Correct Answer: B

Concrete failure feedback drives precise fixes: 'AssertionError: expected {status: 200} got {status: 404} in test_create_user' tells Claude Code exactly what's wrong and where. This is more effective than describing the failure in prose. The test failure IS the specification of what needs to change.

Q58. Your CLAUDE.md references specific file paths for examples (e.g., 'See src/auth/middleware.ts for the authentication pattern'). What maintenance concern does this create?

A) None — file path references are static

B) File paths break when files are moved or renamed, causing Claude Code to look for non-existent examples. Periodically audit CLAUDE.md for stale paths, or prefer describing patterns in text with 'search for AuthMiddleware using Grep' instead.

C) File path references increase CLAUDE.md loading time

D) Absolute paths are not supported in CLAUDE.md

■ Correct Answer: B

Living documentation challenge: CLAUDE.md references can go stale. Options: (1) add CI checks that validate CLAUDE.md file references exist, (2) use grep-based discovery instructions instead of hard paths ('find implementations with Grep using: AuthMiddleware'), (3) regular CLAUDE.md review as part of significant refactors.

Q59. A security-sensitive codebase needs Claude Code to always run static analysis before any code submission. How do you enforce this beyond a system prompt instruction?

A) System prompt instructions are sufficient for this requirement

B) Create a .claude/commands/submit-code.md slash command that explicitly includes the static analysis step in its workflow. This makes the step visible and explicit rather than relying on Claude Code to remember the instruction.

C) Add a Git pre-commit hook that runs static analysis independently

D) Disable Claude Code's write tools and require all changes to go through code review

■ Correct Answer: B

Codified workflows via slash commands: /submit-code runs static analysis, then tests, then commits — the sequence is in the command definition. The pre-commit hook (option C) is also correct as a complementary control but operates outside Claude Code's workflow.

Q60. What is the 'test-then-implement' workflow in Claude Code and when is it most valuable?

A) Running existing tests before making changes to verify the baseline

B) Writing tests first that describe the desired behavior, then asking Claude Code to implement code that makes those tests pass — most valuable for complex business logic where specification via tests is clearer than prose

C) Asking Claude Code to write tests after implementation for regression coverage

D) Using Claude Code to generate test stubs that developers fill in manually

■ Correct Answer: B

Test-driven development with Claude Code: write tests as precise specification, Claude Code implements to green. This works best for complex business logic where 'if the input is X, expect Y' is clearer than prose. Claude Code can then iterate: run tests → see failures → fix implementation → repeat.

ADVANCED QUESTIONS

Questions 61–90

Q61. You are designing a Claude Code configuration for a monorepo with 5 packages: web (React/TS), api (Node/TS), ml-service (Python), infra (Terraform), mobile (React Native). What is the optimal CLAUDE.md architecture?

A) One massive root CLAUDE.md containing all conventions

B) Root CLAUDE.md for shared conventions; package-level CLAUDE.md in each package for language/framework specifics; .claude/rules/ files for cross-cutting concerns (security, testing) with path scoping

C) CLAUDE.md only in packages that have frequent Claude Code usage

D) Separate Claude Code projects for each package

■ Correct Answer: B

Layered architecture: root provides monorepo context and shared rules; packages provide specific context (React conventions, Python typing, Terraform patterns); cross-cutting rules use path scoping (**/*.test.ts for testing conventions, **/*.tf for Terraform). This is maintainable and provides exactly the right context per file.

Q62. A security audit finds that developers are using Claude Code to generate code that bypasses your security controls. The code passes automated tests but violates policy. How do you prevent this at the CLAUDE.md level?

A) Disable Claude Code for security-sensitive modules

B) Define explicit security policies in CLAUDE.md with examples of prohibited patterns: 'Never generate code that bypasses authentication middleware — see src/middleware/auth.ts as the required pattern'

C) Use access controls to prevent Claude Code from reading security files

D) Run all Claude Code output through a separate security review model

■ Correct Answer: B

CLAUDE.md security policies with concrete examples and required patterns are the most effective prevention. Abstract rules ('be secure') are ineffective. Concrete examples of what's prohibited and what the correct pattern looks like provide Claude Code with actionable guidance.

Q63. Your CI pipeline runs 3 parallel Claude Code review jobs (security, performance, correctness) on every PR. Sometimes they produce conflicting recommendations. How do you resolve conflicts architecturally?

A) Run them sequentially so each sees the prior review's output

B) Design each job with explicit scope boundaries in its review criteria; add a conflict resolution step that identifies contradictions and applies a priority order (security > correctness > performance)

C) Use the average of conflicting recommendations

D) Only run one review type per PR to avoid conflicts

■ Correct Answer: B

Separation of concerns with explicit scope prevents most conflicts. When conflicts occur, a conflict resolution step with clear priority order resolves them. Security recommendations override performance suggestions when they conflict. Documenting the priority order makes the system predictable.

Q64. You need Claude Code to consistently use a specific internal logging library instead of console.log/print across all languages (TypeScript, Python, Go). How do you configure this most effectively?

A) Add 'use our logger' to the root CLAUDE.md

B) Create language-specific rules in .claude/rules/ with path scoping, each providing: the banned pattern, the correct import, and a usage example showing the equivalent of console.log in that language's logger API

C) Create a PreToolUse hook in Claude Code that rejects code with console.log

D) This requires a custom model fine-tuned on your codebase

■ Correct Answer: B

Language-specific rules with concrete alternatives: logging.md for Python (paths: ['**/*.py']), logging-ts.md for TypeScript (paths: ['**/*.ts', '**/*.tsx']), each showing the import and equivalent API. Abstract rules without concrete alternatives fail because Claude Code doesn't know your specific logger's API.

Q65. A senior architect wants to use Claude Code to explore three competing architectural approaches for a major system redesign before committing. The codebase is 500K lines. What is the optimal workflow?

A) Run Claude Code three times sequentially on the full codebase

B) Use plan mode to analyze the codebase once (via Explore subagent to avoid context exhaustion); fork three sessions from the analysis result; explore each architectural approach in its own fork; compare results before committing

C) Create three separate git branches and run Claude Code on each

D) Provide the full codebase as context and ask Claude Code to evaluate all three approaches simultaneously

■ Correct Answer: B

Optimal workflow: (1) one Explore analysis (isolated context) produces codebase understanding summary, (2) three forks from that shared understanding baseline, (3) each fork explores one architectural approach without contaminating the others, (4) coordinator compares fork outcomes. This maximizes reuse and isolation.

Q66. Your team's CLAUDE.md has grown to 8,000 tokens and developers report Claude Code sometimes ignoring conventions in the middle of the file. How do you fix this?

A) Increase max_tokens to give Claude Code more capacity

B) Restructure using @import to load only relevant rules, use .claude/rules/ with path scoping so conventions load contextually rather than all at once, and prioritize critical rules at the top

C) Delete less important sections to reduce file size

D) Split into two separate CLAUDE.md files

■ Correct Answer: B

Attention management: very long CLAUDE.md files suffer from middle-content attention dilution. Path-scoped rules load contextually (only when relevant), keeping the active context focused. @import enables modular organization. Critical universal rules stay at the top of root CLAUDE.md where attention is highest.

Q67. How should you design a Claude Code CI workflow for a repository with both application code and infrastructure-as-code (Terraform)?

A) Run the same review job on all file types

B) Separate review jobs with specialized context: application review job (loads CLAUDE.md with coding standards), IaC review job (loads .claude/rules/terraform.md with security and cost guidelines), each triggered by relevant file changes

C) Review only application code — IaC review requires specialized human expertise

D) Combine all reviews into one job to reduce CI cost

■ Correct Answer: B

Specialized context per review type produces better results. Terraform reviews need different criteria (resource naming, state management, cost implications, security groups) than application reviews. Path-triggered jobs (only run Terraform review when .tf files change) reduce unnecessary CI runs.

Q68. A developer reports that a skill they created for codebase analysis is making the main session context unavailable for follow-up questions after the skill runs. What is the likely configuration error and fix?

A) The skill is too long — reduce its length

B) The skill is missing context: fork in its frontmatter; adding it will run the skill in an isolated context, preserving the main session for follow-up

C) The skill should use allowed-tools: [] to prevent context consumption

D) Skills cannot run analysis — use a slash command instead

■ Correct Answer: B

Without context: fork, the skill's verbose output (file reads, analysis, intermediate findings) accumulates in the main session context, crowding out space for subsequent conversation. Adding context: fork isolates all that output.

Q69. You need to implement a Claude Code review system that learns from developer feedback. Developers can mark findings as 'false positive' or 'valid'. How do you incorporate this feedback into future reviews?

A) Manually update CLAUDE.md based on patterns of false positives

B) Maintain a feedback database; before each review, inject aggregated false positive patterns as explicit 'do not flag' examples in the review context; periodically update CLAUDE.md rules based on consistent patterns

C) Adjust temperature based on false positive rate

D) Switch to a different model when false positive rates exceed 20%

■ Correct Answer: B

Feedback loop: store dismissed findings with context → extract patterns → inject as few-shot 'do not report' examples in review prompts → accumulate into permanent CLAUDE.md rules when patterns are consistent. This is continuous improvement without model retraining.

Q70. A large enterprise wants to standardize Claude Code configuration across 50 repositories. Updates to standards must propagate to all repos. What architecture achieves this?

A) Manually update each repo's CLAUDE.md when standards change

B) Maintain a central standards repository; use @import with URLs or git submodule references to include shared standards; repos maintain only project-specific overrides locally

C) Use a single monorepo for all 50 projects

D) Share standards via a shared npm package

■ Correct Answer: B

Centralized standards with local overrides: a shared standards repo contains authoritative rules. Individual repos @import from it (or reference via submodule) and add only project-specific rules locally. Standard updates propagate via dependency update, not manual editing of 50 files.

Q71. A CLAUDE.md has grown organically to 12,000 tokens over 18 months. Claude Code is inconsistently applying rules. What is the refactoring approach?

A) Delete older rules — they're probably outdated anyway

B) Audit rules for current relevance; move subsystem-specific rules to appropriate directory CLAUDE.md files; convert frequently-applicable rules to .claude/rules/ with path scoping; keep only universal rules in root CLAUDE.md

C) Split into two equal halves: CLAUDE.md and CLAUDE2.md

D) Compress the content by removing examples and explanations

■ Correct Answer: B

Audit-driven refactoring: 12,000 tokens is too large for consistent attention. Rules that apply only to src/payments/ should live in src/payments/CLAUDE.md. File-type rules belong in .claude/rules/ with path scoping. Root CLAUDE.md should contain only truly universal rules. Contextual loading > loading everything always.

Q72. A team uses both Terraform and Pulumi in the same repository for different infrastructure components. How do you configure CLAUDE.md to handle both?

A) Choose one and document it exclusively — two IaC tools create confusion

B) Create separate .claude/rules/ files: terraform.md (paths: ['**/*.tf', 'infra/terraform/**']) and pulumi.md (paths: ['infra/pulumi/**', '**/*.ts' and only in infra/]) with each file's conventions, patterns, and gotchas

C) Combine both sets of rules in a single iac.md file

D) Document both tools in root CLAUDE.md with a disambiguation section

■ Correct Answer: B

Path-scoped rules cleanly separate IaC tool conventions. Terraform rules load when editing .tf files; Pulumi rules load when editing Pulumi stacks. Without scoping, Claude Code would receive mixed rules when editing either type, potentially confusing Terraform and Pulumi patterns.

Q73. You are deploying Claude Code as a service where multiple engineers can submit tasks via a web interface. Tasks run asynchronously. What architectural considerations apply?

A) Claude Code is not designed for service deployment — use it only interactively

B) Asynchronous task queue (each task is a Claude Code invocation with -p), job state tracking in a database, result storage and retrieval API, per-user rate limiting, task timeout enforcement, isolated working directories per task, result cleanup policies

C) Use a single shared Claude Code instance for all users

D) Claude Code tasks must be synchronous — implement a polling wait in the web interface

■ Correct Answer: B

Claude Code as a service via -p flag: non-interactive invocations can be queued and executed asynchronously. Isolated working directories prevent cross-task contamination. Job tracking enables status polling. This is a valid production pattern for team-scale Claude Code deployment.

Q74. How do you implement 'drift detection' for CLAUDE.md — detecting when the actual codebase has evolved away from documented conventions?

A) Trust developers to update CLAUDE.md manually

B) Build automated drift detection: CI job that invokes Claude Code to compare CLAUDE.md's claims against the actual codebase (e.g., 'CLAUDE.md says we use X pattern — check if all relevant files follow it'); report discrepancies as CLAUDE.md maintenance issues

C) Re-generate CLAUDE.md from scratch monthly using Claude Code

D) Drift is acceptable — CLAUDE.md is aspirational documentation

■ Correct Answer: B

Automated drift detection: Claude Code + CI can check if CLAUDE.md claims match reality. 'CLAUDE.md says all API responses use {data, error, meta} envelope — scan all response-generating files for compliance.' Discrepancies flagged as maintenance issues. CLAUDE.md drift makes it misleading rather than helpful.

Q75. A large enterprise wants Claude Code configuration to be centrally managed with department-level overrides and project-level customizations. What is the most scalable architecture?

A) One shared CLAUDE.md for all teams

B) Three-tier CLAUDE.md hierarchy: Enterprise-level baseline (git submodule or package in all repos), department-level overrides (@import in team repos), project-level customizations (local CLAUDE.md that @imports department config). Central governance reviews and approves enterprise and department levels.

C) Deploy a Claude Code configuration API that serves CLAUDE.md content dynamically

D) Have each project team manage their own CLAUDE.md independently

■ Correct Answer: B

Three-tier governance: enterprise (universal standards, security, compliance), department (team-specific patterns, tools), project (feature-specific context). Git submodule or package for enterprise tier enables centralized updates that propagate to all repos via dependency update. This balances standardization with autonomy.

Q76. How do you implement Claude Code workflows that span multiple repositories — for example, updating an API in one repo and its corresponding clients in three other repos?

A) Cross-repo workflows are not possible with Claude Code

B) Use a coordinator script that clones all affected repos into a workspace, invokes Claude Code with a task context that spans all repos, uses Bash tools for cross-repo git operations, and opens PRs in all affected repos

C) Update repos sequentially — Claude Code handles one repo at a time

D) Share a single CLAUDE.md across all repos via symlink

■ Correct Answer: B

Cross-repo coordinator: a script orchestrates multi-repo changes by providing Claude Code with a workspace containing all affected repos, a task description of the breaking change and its impact, and instructions to update each repo. Bash tools enable git operations. PR creation can be scripted. Sequential updates also work but miss cross-repo consistency checking.

Q77. Your team wants to use Claude Code for code generation but is concerned about intellectual property — specifically, generated code that might be derived from training data that includes open-source code with incompatible licenses. How do you address this?

A) All Claude-generated code is original — no IP concerns

B) Implement a code provenance workflow: scan generated code with license-detection tools (FOSSA, SPDX), require Claude Code to explain algorithmic choices for non-trivial implementations, establish code review policies that evaluate generated code like any vendor contribution

C) Only use Claude Code for boilerplate — never for algorithm implementation

D) Require all generated code to include a 'generated by Claude' comment

■ Correct Answer: B

IP risk management for AI-generated code: provenance scanning detects copied code patterns, explanations reveal if the model is drawing from specific implementations vs. general knowledge, code review policies treat AI output like third-party code. This is the responsible enterprise approach to AI-generated code.

Q78. You need Claude Code to work with a proprietary internal framework that post-dates its training data. The framework has 300 files of documentation. How do you make Claude Code effective with this framework?

A) Provide all 300 documentation files in the CLAUDE.md via @import — more context is better

B) Create a curated CLAUDE.md section for the framework: core concepts, key APIs with examples, common patterns, and gotchas. Add a MCP resource exposing the full documentation for on-demand lookup. Let Claude Code discover details as needed.

C) Claude Code cannot learn new frameworks — wait for a new model version

D) Provide the full documentation as context in every prompt

■ Correct Answer: B

Curated core knowledge + on-demand deep dive: CLAUDE.md provides the 80% common case (framework concepts, typical API usage, patterns). For edge cases, MCP resource access to full documentation enables targeted lookup. Loading 300 files into every session is wasteful — most sessions won't need all documentation.

Q79. How do you design a Claude Code workflow that handles merge conflicts when two Claude Code sessions have independently modified the same files?

A) Configure Claude Code sessions to never modify the same files

B) Implement a conflict resolution workflow: create a merge branch, present both versions to a new Claude Code session with conflict context (why each version was created), ask it to produce a semantically correct merge that preserves both intents

C) Always apply the most recent Claude Code session's changes

D) Route all changes through a single serial Claude Code session

■ Correct Answer: B

Semantic merge via Claude Code: present conflict context (Branch A's intent, Branch B's intent) to a fresh Claude Code session specialized in conflict resolution. It understands both changes semantically and produces a merge that satisfies both intents. This outperforms git's line-based merge for logic conflicts.

Q80. A security audit reveals that Claude Code occasionally uses deprecated npm packages with known vulnerabilities in its generated code. How do you prevent this systematically?

A) Add 'do not use deprecated packages' to CLAUDE.md

B) CLAUDE.md documents the approved package list with versions; a CI step after Claude Code runs npm audit on generated package.json changes; Claude Code is instructed to check CLAUDE.md's approved packages before adding new dependencies

C) Disable Claude Code's ability to modify package.json

D) Manually review all package.json changes before committing

■ Correct Answer: B

Approved package list + automated audit: CLAUDE.md provides positive guidance (use these approved packages), CI provides the safety net (npm audit on every change). Claude Code checking the approved list reduces violations. CI catches anything that slips through. Defense in depth for dependency security.

Q81. You are building a Claude Code-powered pair programming tool for a coding education platform. Students learn best by making mistakes. How do you configure Claude Code to teach rather than solve?

A) Disable Claude Code for education — students should not use AI

B) Configure CLAUDE.md with Socratic teaching mode: Claude Code asks guiding questions instead of providing solutions, points to relevant documentation rather than copying it, shows partial examples with key parts missing, provides hints at 3 levels of specificity before the full answer

C) Claude Code cannot be configured for teaching — it always provides complete solutions

D) Use lower max_tokens to force shorter, less complete answers

■ Correct Answer: B

Teaching-mode CLAUDE.md: explicit instructions on pedagogical approach. 'When a student asks how to fix a bug: first ask what they think is wrong, then point to the relevant documentation section, then show a similar example from a different context, only provide the direct fix if they remain stuck after three hints.' CLAUDE.md fully controls this behavior.

Q82. How do you implement 'reproducible Claude Code runs' — ensuring that re-running the same task produces comparable results for testing and validation?

A) Reproducibility is impossible with LLMs — accept non-determinism

B) Maximize determinism through: fixed model versions (not 'latest'), low temperature, structured output schemas (tool use), defined acceptance criteria in CLAUDE.md, version-controlled task definitions. Test against pass/fail criteria rather than exact output matching.

C) Use temperature=0 for perfect reproducibility

D) Record and replay API responses for deterministic testing

■ Correct Answer: B

Practical reproducibility: exact output reproducibility is infeasible with LLMs. Functional reproducibility (passes defined acceptance criteria) is achievable. Fixed model versions prevent model-change regressions. Structured output schemas narrow variance. Criteria-based testing is robust to acceptable output variation.

Q83. How should CLAUDE.md evolve as a project grows from a startup (10 files) to an enterprise product (10,000 files)?

A) CLAUDE.md should stay concise regardless of project size

B) Evolve from: monolithic single file (startup) → modular with @import (growth) → layered hierarchy with path-scoped rules (enterprise). Introduce MCP resources for on-demand documentation access. Regularly audit for sections that are no longer relevant.

C) Create separate CLAUDE.md files for each feature as the project grows

D) Enterprise projects should rely on code conventions only — not CLAUDE.md

■ Correct Answer: B

CLAUDE.md evolution mirrors project maturity: startup needs simple context; growth needs organization (@import for modular maintenance); enterprise needs hierarchy (global + team + project) with path scoping. Regular audits prevent CLAUDE.md from becoming a graveyard of outdated context.

Q84. You need to implement a Claude Code workflow for data migration — extracting data from a legacy database, transforming it, and loading into a new schema. The migration affects 50 million records. What safeguards must the workflow include?

A) Run the migration directly with Claude Code on the production database

B) Workflow safeguards: (1) develop and test against a data sample (1,000 records), (2) dry-run mode that validates transformation without writing, (3) incremental processing with checkpoints, (4) rollback capability via backup or reversible write, (5) record-count validation at each stage, (6) human review of transformation logic before production run

C) Trust Claude Code's data migration code — it's been validated

D) Use only SQL migrations — Claude Code is not appropriate for data migration

■ Correct Answer: B

Data migration safety: sample-first development catches issues on small data, dry-run validates transformation logic, incremental + checkpoints enable recovery from mid-migration failures, rollback capability for disaster recovery, record-count validation catches data loss, human review of transformation logic before production. 50M records is too high-stakes for untested code.

Q85. A DevOps team wants to use Claude Code for infrastructure provisioning with Terraform. How should the CLAUDE.md and skills be configured for this use case?

A) Infrastructure is too sensitive for Claude Code automation

B) CLAUDE.md: Terraform conventions, state backend config, tagging standards, module structure. Skills: plan-and-review skill (runs terraform plan, presents output for human review), apply-with-approval skill (requires explicit confirmation, logs all changes). Allowed tools: Bash (limited), Read, Write. Excluded: any tool that bypasses the plan/apply cycle.

C) Give Claude Code full AWS permissions and let it provision directly

D) Use Claude Code only for Terraform code review, never for execution

■ Correct Answer: B

IaC automation with Claude Code: CLAUDE.md provides Terraform-specific context, skills enforce the required workflow (plan → human review → apply), tool restrictions prevent bypassing the safety cycle. The human review gate in the apply skill is the critical safety control for infrastructure changes.

Q86. How do you design a Claude Code-based code archaeology system that answers questions like 'why was this architectural decision made?' using git history and code comments?

A) This is beyond Claude Code's capabilities — use a dedicated code archaeology tool

B) Implement a code archaeology skill: uses Bash (git log, git blame, git show), Grep (for related comments/issues), and Read to trace decision history. Returns structured findings: decision_point, evidence (commit messages, PR references, comments), likely_rationale, confidence. Runs in isolated context (context: fork) to avoid polluting main session.

C) Ask developers to document all decisions in CLAUDE.md

D) Use semantic code search tools only — git history is too noisy

■ Correct Answer: B

Code archaeology via Claude Code skill: git history access (via Bash), comment/issue mining (via Grep), file version comparison (via Bash git show) combine to reconstruct decision context. Structured output (decision, evidence, rationale) makes findings actionable. Context: fork keeps the verbose investigation isolated from the main session.

Q87. How should you handle the case where Claude Code produces code that passes all tests but violates an implicit architectural constraint not documented in CLAUDE.md (such as circular dependency introduction)?

A) Accept the code — tests passing is sufficient

B) (1) Run dependency graph analysis after every significant Claude Code session. (2) Add the violated constraint to CLAUDE.md immediately: 'No circular dependencies — run tsc --noEmit and madge --circular src/ to verify before committing.' (3) Create a CI check that enforces the constraint automatically.

C) Reject all Claude Code output that isn't manually reviewed

D) Update the test suite to catch architectural violations

■ Correct Answer: B

Discovery-to-documentation loop: architectural violations discovered via Claude Code should immediately become documented CLAUDE.md constraints + automated checks. This converts implicit knowledge into explicit guidance. Option D (test suite) is also valuable but CLAUDE.md + CI check prevents recurrence more comprehensively.

Q88. You are implementing a Claude Code workflow for a financial services company where code must be reviewed by compliance before deployment. How do you integrate compliance review into the Claude Code workflow?

A) Compliance review happens after Claude Code finishes — it's a separate process

B) Design a compliance-integrated workflow: Claude Code generates code + auto-generates a compliance questionnaire based on the changes (data types touched, regulatory categories affected). Questionnaire is submitted to compliance team. Claude Code provides supporting analysis (what regulations apply, why the implementation meets them). Human compliance sign-off gates the PR.

C) Configure Claude Code to only generate compliant code via system prompt

D) Compliance review is unnecessary for AI-generated code

■ Correct Answer: B

Compliance-integrated workflow: Claude Code doesn't just generate code — it also generates the compliance evidence package (questionnaire, regulatory mapping, implementation rationale). This reduces the compliance team's work while maintaining oversight. Human sign-off remains the gate. System prompt compliance instructions are insufficient for regulatory requirements.

Q89. How do you implement 'skill composition' in Claude Code — where one complex skill invokes simpler component skills?

A) Skill composition is not supported — each skill runs independently

B) A high-level skill's prompt can reference and invoke lower-level skills explicitly: 'First run /analyze-dependencies, then /check-security-issues, then synthesize both results.' Claude Code chains the skills, passing outputs between them as context.

C) Combine all component skill logic into one large skill file

D) Use a Bash script to chain skill invocations externally

■ Correct Answer: B

Skill composition via explicit invocation: a meta-skill can reference component skills. Claude Code executes them in sequence, using each skill's output as context for the next. This enables complex workflows built from simple, independently tested components. Combining into one file loses the modularity and independent testability.

Q90. A team wants Claude Code to automatically update their API documentation (OpenAPI spec) whenever it modifies API endpoint handlers. How do you implement this as a persistent workflow?

A) Documentation updates require separate manual steps

B) Path-scoped skill: a .claude/rules/api-endpoints.md with paths: ['src/routes/**/*.ts'] that instructs Claude Code to always update the OpenAPI spec when modifying routes. The rule specifies the spec location and the update methodology. CI validates spec-code consistency.

C) Use a separate documentation generation tool

D) Add 'always update OpenAPI spec' to CLAUDE.md globally

■ Correct Answer: B

Path-scoped documentation rules: the rule loads only when editing route files, making spec update automatic for that context. CI validation catches cases where the rule was bypassed. Global CLAUDE.md instruction applies to all files — scope it to where it's relevant to avoid noise.