

Prompt, Context & Token Discipline

Implementation Playbook

20 ValidTec Track 2 source entries organized into five practical chapters - four entries per chapter - for building reliable prompts, cleaner context, controlled token use, project memory, and measurable tester-AI workflows.

Track 2 focus

Separate durable policy from task requests. Supply only the context needed for the decision. Version prompts and memory, verify outputs against known cases, and improve the workflow through measured evidence.

About This Playbook

This book is a practical implementation companion for ValidTec Track 2: Prompt, Context & Token Discipline. It organizes 20 source entries into five stages that a software testing engineer can execute in sequence.

Important note

The chapters synthesize the themes represented by the source article titles into an original implementation framework. Readers should use the original ValidTec articles for the full article-specific discussion, examples, and references. The current Track 2 page lists one loop-engineering article twice; its second appearance is used as an operating-governance lens in Chapter 5.

How to use it

1. Choose one real QA workflow with a measurable pain point.
2. Complete Chapter 1 to define stable prompt policy, task instructions, and output contracts.
3. Complete Chapter 2 to select authoritative context and set an evidence-based token budget.
4. Use Chapter 3 to build concise project memory and reusable testing skills.
5. Use Chapter 4 to establish the tester-AI operating workflow and human review points.
6. Use Chapter 5 to version, evaluate, promote, and roll back prompt-system changes.
7. Run the 90-day roadmap and keep the artifacts as portfolio evidence.

Track 2 completion standard

- System policy, task instructions, context, memory, and output contracts have explicit boundaries.
- AI outputs cite evidence, expose unknowns, and pass deterministic or human review.
- Token reduction is accepted only when quality remains at or above the approved baseline.
- Prompt and memory changes are versioned, evaluated, approved, and reversible.
- The workflow has measurable targets for coverage, accuracy, latency, cost, and human correction effort.

Contents

Chapter 1: Build a Prompt Operating Standard

System policy, task briefs, output contracts, examples, and prompt review.

Chapter 2: Engineer Context and Control Token Cost

Source authority, context packets, token budgets, compact formats, and citations.

Chapter 3: Create Project Memory and Reusable Skills

CLAUDE.md, testing commands, conventions, known issues, skills, and maintenance.

Chapter 4: Run a Tester-AI Delivery Workflow

Feature intake, risk analysis, test design, automation, execution, and release evidence.

Chapter 5: Establish the Prompt Improvement Loop

Evaluation cases, telemetry, failure classification, promotion, and rollback.

Track 2 Reference Architecture

A compact mental model connecting prompt, context, memory, workflow, and improvement layers.

90-Day Roadmap: From prompt baseline to controlled tester-AI workflow

A staged implementation plan and reusable artifact checklist.

Article Index: All 20 Track 2 source entries

A reference map showing where each article entry appears.

What Completion Looks Like

The final implementation standard and next improvements.

Build a Prompt Operating Standard

Separate durable policy from task-specific instructions and make every prompt reviewable.

Chapter outcome
A Prompt Operating Standard containing a stable system prompt, a task brief, an output contract, a review checklist, and a small approved prompt library.

This chapter converts four related Track 2 source entries into one implementation playbook. Complete the exit criteria before moving to the next chapter.

The four source entries

#	Source article	Role in this chapter
7	Prompting Is Now a Core Skill for Software Test Engineers (2026+)	Treat prompting as an engineering skill with repeatable quality criteria.
2	How System vs. User Prompts Change the Game in AI-Powered Software Testing	Separate strategy and guardrails from the immediate testing task.
1	Make Your Testing Agents Communicate Like a Team: Verbalized Sampling Prompts	Make options, trade-offs, choice, and rationale visible in agent handoffs.
18	7 Claude Code Prompts That Help Software Testing Engineers Work Like Experienced Engineers	Turn common testing activities into reusable, senior-level prompt patterns.

Four-phase playbook

Phase	Action	Output	Gate
1. Baseline	Select one recurring QA task and save three current outputs.	Baseline set	Variation is visible
2. Separate	Move durable rules into the system layer and task facts into the user layer.	Two-layer prompt	No mixed concerns
3. Contract	Define the required fields, format, unknown handling, and evidence.	Output schema	Output is parseable
4. Review	Score coverage, correctness, relevance, and actionability.	Prompt scorecard	Target score reached

Operating principles

- Use the system prompt for role, quality policy, boundaries, definitions, and the output contract.
- Use the user prompt for the feature, change, environment, acceptance criteria, risks, and requested artifact.
- Require the model to label assumptions and unknowns instead of inventing missing facts.
- Prefer a small prompt library with owners and tests over a large folder of unreviewed examples.

Implementation steps

1. Choose the QA task

Choose a high-frequency task such as test design, defect triage, change-impact analysis, or release reporting.

2. Write the success statement

Write a one-sentence objective and a measurable definition of done.

3. Establish the system layer

Create a stable instruction layer: testing role, heuristics, severity definitions, safety rules, and output fields.

4. Create the task brief

Create a task brief template: scope, inputs, constraints, environment, risks, references, and exclusions.

5. Add approved examples

Add one or two approved examples only when they materially reduce ambiguity.

6. Baseline and approve

Run the same three cases twice and compare completeness, consistency, and unsupported claims.

Artifacts to keep

- System prompt v1
- Task brief template
- Output schema
- Prompt scorecard
- Approved prompt library

Exit criteria

- [] Durable rules and task facts are separated.

- [] Outputs follow the defined structure.
- [] Unknowns and assumptions are explicit.
- [] At least three known cases meet the quality target.
- [] A tester owns changes to the prompt standard.

Chapter completion rule

Do not advance because the design sounds impressive. Advance when the artifacts exist, the exit criteria can be demonstrated, and the workflow can be executed against known cases.

Implementation worksheet

Capture the decisions for this chapter before implementation moves forward.

Workflow / use case	
Primary owner and reviewer	
Metric + target	
Largest risk + control	
Evidence / artifact to retain	

Engineer Context and Control Token Cost

Give the model the smallest sufficient evidence set, in a structure it can use reliably.

Chapter outcome

A Context Contract that defines approved sources, selection rules, ordering, freshness, token budget, compression, and citations for one testing workflow.

This chapter converts four related Track 2 source entries into one implementation playbook. Complete the exit criteria before moving to the next chapter.

The four source entries

#	Source article	Role in this chapter
4	Context Engineering for Software Testing: A Practical Playbook (Automation -> Manual -> LLM/SLM -> Copilots)	Treat context selection and structure as part of test design.
3	TOON vs JSON: Cut Tokens, Speed Up Prompts - with Verbalized Sampling, SLMs, and Google's RAG API	Compare compact boundary formats when structured payloads dominate the prompt.
14	How Understanding 10 LLM Engineering Concepts Helps Software Testing Engineers in the SDLC Workflow	Apply LLM engineering concepts to retrieval, memory, evaluation, and SDLC handoffs.
9	AI Tools Must Build the 'Physics' Into the Product, Not Leave It to the User	Make goal setting, constraints, phases, checkpoints, and verification part of the workflow.

Four-phase playbook

Phase	Action	Output	Gate
1. Inventory	List candidate facts, files, examples, and historical evidence.	Source inventory	Each source has an owner
2. Select	Rank evidence by relevance, authority, freshness, and risk.	Selection rules	Noise is excluded
3. Compress	Summarize or normalize repetitive material without losing decision facts.	Context packet	Budget is met
4. Prove	Require source references and compare results against a full-context baseline.	Context evaluation	Quality is preserved

Operating principles

- Context quality matters more than context volume; irrelevant history can lower accuracy.
- Place the most authoritative task facts close to the request and distinguish evidence from commentary.
- Keep upstream JSON contracts when systems depend on them; optimize the model boundary only after measurement.
- Track token input, latency, cost, omissions, and unsupported claims together - savings alone are not success.

Implementation steps

1. Inventory required evidence

List the minimum evidence needed to decide: ticket, acceptance criteria, diff, API schema, logs, prior defects, and test inventory.

2. Rank authority and freshness

Assign authority and freshness rules; for example, an approved specification outranks a chat summary.

3. Assemble the context packet

Create a context packet with labeled sections: task, product rules, change evidence, test evidence, constraints, and unknowns.

4. Set the token budget

Set a hard token budget and a drop order for optional material.

5. Benchmark compact formats

Benchmark JSON against a compact format only on representative payloads and validate lossless reconstruction.

6. Require source citations

Add citations or source IDs to every material recommendation so testers can verify the reasoning.

Artifacts to keep

- Source inventory
- Authority/freshness matrix
- Context packet template
- Token benchmark
- Citation rules

Exit criteria

- [] Every included source has a stated purpose.

- [] The packet fits the token budget.
- [] Important facts survive compression.
- [] Recommendations cite source IDs.
- [] Reduced context performs no worse than the approved baseline.

Chapter completion rule

Do not advance because the design sounds impressive. Advance when the artifacts exist, the exit criteria can be demonstrated, and the workflow can be executed against known cases.

Implementation worksheet

Capture the decisions for this chapter before implementation moves forward.

Workflow / use case	
Primary owner and reviewer	
Metric + target	
Largest risk + control	
Evidence / artifact to retain	

Create Project Memory and Reusable Skills

Turn recurring testing knowledge into maintained instructions instead of repeated conversation.

Chapter outcome
A governed project-memory package with a concise CLAUDE.md, modular testing skills, known-issue records, maintenance ownership, and a retirement process.

This chapter converts four related Track 2 source entries into one implementation playbook. Complete the exit criteria before moving to the next chapter.

The four source entries

#	Source article	Role in this chapter
5	CLAUDE.md Files - The Missing Manual (Intro, How-To, and Pro Tips)	Structure CLAUDE.md as durable project operating instructions.
10	Creating Project Memory in Claude Code to Increase Productivity for Software Testing Engineers	Capture recurring commands, environments, test-data rules, and defect patterns.
11	How Claude Code Project Memory and Skills Can 10x a Software Testing Engineer's Productivity	Separate always-needed memory from task-specific reusable skills.
12	How AutoDream Makes Software Test Engineers More Effective Across Claude Chat, Cowork, and Code	Evaluate complementary memory approaches by workflow need, transparency, and control.

Four-phase playbook

Phase	Action	Output	Gate
1. Capture	Record only knowledge repeatedly explained to people or AI tools.	Memory backlog	Reuse value is clear
2. Structure	Place stable basics in project memory and detailed procedures in skills.	Memory map	No giant instruction file
3. Validate	Test the memory against common tasks and deliberate counterexamples.	Memory tests	Rules are followed
4. Maintain	Assign owners, review dates, and deletion rules.	Governance log	Stale guidance is removed

Operating principles

- Memory is an operational control, not a notebook; store instructions that should change future behavior.
- Keep project memory short enough to load routinely. Move verbose workflows and examples into named skills or references.
- Never store credentials, production data, personal data, or secrets in AI-readable project instructions.
- Record rejected tools and patterns when the reason remains valid, so the assistant does not repeat discarded advice.

Implementation steps

1. Capture repeated knowledge

Start with project purpose, critical user flows, supported environments, and the approved test stack.

2. Create the project memory

Add exact commands for smoke, API, UI, regression, linting, and report generation.

3. Document safety boundaries

Document test data rules, authentication setup, cleanup requirements, and production-safety restrictions.

4. Add known issues

Create a known-issues section with symptoms, verified causes, workarounds, and expiry dates.

5. Create reusable testing skills

Move multi-step procedures into modular skills such as defect reproduction, API test generation, or release-summary review.

6. Establish the review cycle

Review memory after tool, architecture, environment, or team-policy changes.

Artifacts to keep

- CLAUDE.md
- Testing skill catalog
- Known-issue register
- Memory test cases
- Review/retirement log

Exit criteria

- [] The file contains no secrets or sensitive data.

- [] Routine commands and conventions are accurate.
- [] Long procedures live outside core memory.
- [] Known issues have owners and expiry dates.
- [] Memory changes are versioned and reviewed.

Chapter completion rule

Do not advance because the design sounds impressive. Advance when the artifacts exist, the exit criteria can be demonstrated, and the workflow can be executed against known cases.

Implementation worksheet

Capture the decisions for this chapter before implementation moves forward.

Workflow / use case	
Primary owner and reviewer	
Metric + target	
Largest risk + control	
Evidence / artifact to retain	

Run a Tester-AI Delivery Workflow

Choose the right Claude surface, control the work in phases, and preserve human review.

Chapter outcome

A phase-gated tester-AI workflow that moves from feature intake through risk analysis, test design, automation, execution, reporting, and release evidence.

This chapter converts four related Track 2 source entries into one implementation playbook. Complete the exit criteria before moving to the next chapter.

The four source entries

#	Source article	Role in this chapter
8	How Claude Cowork + Claude Code Can 2-5x a Software Test Engineer's Effectiveness	Use conversational, collaborative, and code-execution surfaces for the tasks they handle best.
13	AutoDream vs Project Memory in Claude: Chat, Cowork, and Code	Choose between assisted exploration and persistent project context based on workflow needs.
6	Top 6 AI Skills for 2026 That Will Boost Software Testing Engineer Effectiveness	Develop the AI literacy needed to direct, evaluate, and improve testing work.
20	From New Feature to Release Confidence: Practical AI Prompts for Software Testers	Connect feature prompts, regression prompts, automation prompts, and reporting prompts to release confidence.

Four-phase playbook

Phase	Action	Output	Gate
1. Frame	Clarify feature, users, change, constraints, risks, and acceptance criteria.	Feature brief	Scope approved
2. Design	Generate and review risk-based scenarios, data, and expected results.	Test design	Coverage approved
3. Build	Create or update automation using project standards and minimal changes.	Test change	Code reviewed
4. Prove	Run, triage, summarize evidence, and state remaining risk.	Release packet	Decision is supported

Operating principles

- Use chat for framing, explanation, and critique; use collaborative workspaces for document-heavy synthesis; use code tools for repository-grounded changes.
- Separate planning, implementation, debugging, review, and handoff so each phase has one definition of done.
- Do not let generated test quantity substitute for risk coverage, accurate oracles, maintainability, and execution evidence.
- A human tester remains accountable for scope, expected behavior, destructive actions, and the release recommendation.

Implementation steps

1. Frame the feature

Turn the feature ticket into a test brief listing changed behavior, dependencies, data, roles, platforms, and risks.

2. Generate the risk plan

Ask for multiple risk-based test approaches, then select and justify the smallest sufficient plan.

3. Trace the proposed tests

Map every proposed test to an acceptance criterion, failure mode, historical defect, or operational risk.

4. Create the automation change

Generate automation only after framework conventions, fixtures, selectors, and cleanup rules are in context.

5. Execute the justified regression

Run focused tests first, then the justified regression set; quarantine flakes only with evidence and an owner.

6. Produce the release packet

Produce a release packet with results, failures, coverage gaps, assumptions, residual risks, and recommendation.

Artifacts to keep

- Feature test brief
- Risk/coverage map
- Reviewed automation change
- Execution evidence
- Release recommendation

Exit criteria

[] Every critical risk maps to a test or accepted gap.

- [] Generated code follows project standards.
- [] Execution results are reproducible.
- [] Failures and flakes have dispositions.
- [] The release recommendation states residual risk.

Chapter completion rule

Do not advance because the design sounds impressive. Advance when the artifacts exist, the exit criteria can be demonstrated, and the workflow can be executed against known cases.

Implementation worksheet

Capture the decisions for this chapter before implementation moves forward.

Workflow / use case	
Primary owner and reviewer	
Metric + target	
Largest risk + control	
Evidence / artifact to retain	

Establish the Prompt Improvement Loop

Treat prompts, context, memory, and workflow rules as testable assets that improve through evidence.

Chapter outcome

A PromptOps loop with versioned assets, evaluation cases, run telemetry, review thresholds, failure taxonomy, and controlled promotion or rollback.

This chapter converts four related Track 2 source entries into one implementation playbook. Complete the exit criteria before moving to the next chapter.

The four source entries

#	Source article	Role in this chapter
15	Patterns That Help Software Test Engineers Improve Daily Productivity by 5-10x	Combine repeatable productivity patterns instead of depending on one-off clever prompts.
16	Claude Code, Chat, and Cowork: How Loop Engineering Helps Software Testing Engineers Get Better Results	Use Claude Chat, Cowork, and Code as stages in a deliberate learning loop.
17	Loop Engineering for Software Testing Engineers: Building AI Systems That Improve Through Repetition	Build AI-assisted testing systems that improve through repeated evaluation and correction.
19	Claude Code, Chat, and Cowork: How Loop Engineering Helps Software Testing Engineers Get Better Results	Apply the loop-engineering article again as an operating and governance lens.

Four-phase playbook

Phase	Action	Output	Gate
1. Observe	Capture task, asset version, context set, output, review, cost, and timing.	Run record	Evidence is complete
2. Classify	Label failures by prompt, context, memory, model, tool, or evaluator cause.	Failure taxonomy	Cause is testable
3. Change	Modify one controlled variable and rerun the evaluation set.	Candidate version	Regression is measured
4. Promote	Approve, canary, monitor, and roll back when thresholds fail.	Release record	Quality remains stable

Operating principles

- Version prompts, context selectors, schemas, examples, and memory together with clear ownership.
- Change one material variable at a time when diagnosing failures; otherwise improvement claims are weak.
- Use a fixed evaluation set plus recent production-like cases so the loop protects both stability and relevance.
- Reward verified QA outcomes: better coverage, fewer unsupported claims, lower correction effort, and faster useful completion.

Implementation steps

1. Build the evaluation set

Create 10-20 representative cases: normal, boundary, negative, ambiguous, missing-context, and adversarial examples.

2. Define the scoring rubric

Define scoring rubrics for factual support, risk coverage, output validity, actionability, and human correction effort.

3. Capture run telemetry

Log prompt version, memory version, context IDs, model/settings, latency, cost, validator results, and reviewer decision.

4. Classify failures

Classify each failure before changing the asset; distinguish bad instructions from missing evidence or faulty tools.

5. Run regression comparisons

Run regression comparisons and block promotion when a critical case worsens even if the average score rises.

6. Promote and roll back safely

Promote through review, a limited canary, monitoring, and an immediate rollback path.

Artifacts to keep

- Evaluation case set
- Scoring rubric
- Versioned prompt registry
- Run telemetry
- Promotion/rollback log

Exit criteria

[] Critical tasks have representative evaluation cases.

- [] Every run is tied to asset versions.
- [] Failures use a shared taxonomy.
- [] Promotion thresholds are explicit.
- [] A rollback restores the last approved behavior.

Chapter completion rule

Do not advance because the design sounds impressive. Advance when the artifacts exist, the exit criteria can be demonstrated, and the workflow can be executed against known cases.

Implementation worksheet

Capture the decisions for this chapter before implementation moves forward.

Workflow / use case	
Primary owner and reviewer	
Metric + target	
Largest risk + control	
Evidence / artifact to retain	

Track 2 Reference Architecture

Use this as a compact mental model for the complete playbook. Each layer corresponds to implementation work completed in one or more chapters.

5. Improve	Evaluation set Failure taxonomy Run telemetry Promotion gate Rollback log
4. Operate	Feature brief Risk map Automation review Execution evidence Release packet
3. Remember	CLAUDE.md Testing skills Known issues Commands Owners Review dates
2. Context	Source authority Freshness Context packet Token budget Source IDs Citations
1. Prompt	System policy Task brief Output schema Approved examples Quality scorecard

Golden-path run

1. Receive a bounded QA task and identify the decision or artifact required.
2. Apply the approved system policy and task-brief template.
3. Retrieve only authoritative, current context required for the task.
4. Use maintained project memory and a named testing skill when applicable.
5. Plan the work, expose assumptions, and pause at defined human-review points.
6. Create the testing artifact and validate structure, evidence, safety, and risk coverage.
7. Persist versions, evaluation results, timing, cost, reviewer decision, and final output.
8. Convert meaningful failures and human corrections into permanent evaluation cases.

90-Day Track 2 Implementation Roadmap

Window	Focus	Work	Exit
Days 1-15	Prompt baseline	Choose one QA workflow; collect baseline examples; define the system layer, task brief, schema, and metrics.	Prompt standard approved; known cases assembled.
Days 16-30	Context discipline	Inventory sources; define authority and freshness; create context packet and token budget; run comparison.	Reduced context preserves approved quality.
Days 31-45	Memory and skills	Create concise CLAUDE.md; add commands, conventions, data rules, known issues, skills, owners, and expiry dates.	Memory and skills pass validation tasks.
Days 46-60	Working workflow	Connect feature intake, risk analysis, test design, automation review, execution, and release evidence.	Workflow passes initial golden set.
Days 61-75	Evaluation and telemetry	Add evaluation cases, rubric, asset versions, run records, failure taxonomy, thresholds, and rollback.	Failures are diagnosable and comparable.
Days 76-90	Controlled rollout	Pilot on real QA work; retain human approvals; measure quality, coverage, corrections, latency, and cost.	Documented go/no-go decision for broader use.

Minimum portfolio package

- Prompt Operating Standard with the system layer, task brief, schema, and quality scorecard.
- Context Contract showing source authority, freshness, token budget, and citations.
- Sanitized project memory and one reusable testing skill.
- Evaluation set and report covering normal, boundary, negative, missing-context, and adversarial cases.
- Run record showing asset versions, validators, human review, timing, cost, and final status.
- Before/after metrics comparing quality, risk coverage, correction effort, latency, and cost.
- One retrospective explaining failures, changes made, and new regression cases.

Portfolio framing

Show engineering discipline, not clever wording. Employers and reviewers should be able to see the problem, controls, evidence selection, repeatability, test results, and measured outcome.

Track 2 Article Index

All 20 source entries supplied for this Track 2 book, grouped four per implementation chapter.

Chapter	Article	Playbook stage
Chapter 1	Prompting Is Now a Core Skill for Software Test Engineers (2026+)	Build a Prompt Operating Standard
Chapter 1	How System vs. User Prompts Change the Game in AI-Powered Software Testing	Build a Prompt Operating Standard
Chapter 1	Make Your Testing Agents Communicate Like a Team: Verbalized Sampling Prompts	Build a Prompt Operating Standard
Chapter 1	7 Claude Code Prompts That Help Software Testing Engineers Work Like Experienced Engineers	Build a Prompt Operating Standard
Chapter 2	Context Engineering for Software Testing: A Practical Playbook (Automation -> Manual -> LLM/SLM -> Copilots)	Engineer Context and Control Token Cost
Chapter 2	TOON vs JSON: Cut Tokens, Speed Up Prompts - with Verbalized Sampling, SLMs, and Google's RAG API	Engineer Context and Control Token Cost
Chapter 2	How Understanding 10 LLM Engineering Concepts Helps Software Testing Engineers in the SDLC Workflow	Engineer Context and Control Token Cost
Chapter 2	AI Tools Must Build the 'Physics' Into the Product, Not Leave It to the User	Engineer Context and Control Token Cost
Chapter 3	CLAUDE.md Files - The Missing Manual (Intro, How-To, and Pro Tips)	Create Project Memory and Reusable Skills
Chapter 3	Creating Project Memory in Claude Code to Increase Productivity for Software Testing Engineers	Create Project Memory and Reusable Skills
Chapter 3	How Claude Code Project Memory and Skills Can 10x a Software Testing Engineer's Productivity	Create Project Memory and Reusable Skills
Chapter 3	How AutoDream Makes Software Test Engineers More Effective Across Claude Chat, Cowork, and Code	Create Project Memory and Reusable Skills
Chapter 4	How Claude Cowork + Claude Code Can 2-5x a Software Test Engineer's Effectiveness	Run a Tester-AI Delivery Workflow
Chapter 4	AutoDream vs Project Memory in Claude: Chat, Cowork, and Code	Run a Tester-AI Delivery Workflow
Chapter 4	Top 6 AI Skills for 2026 That Will Boost Software Testing Engineer Effectiveness	Run a Tester-AI Delivery Workflow

Chapter	Article	Playbook stage
Chapter 4	From New Feature to Release Confidence: Practical AI Prompts for Software Testers	Run a Tester-AI Delivery Workflow
Chapter 5	Patterns That Help Software Test Engineers Improve Daily Productivity by 5-10x	Establish the Prompt Improvement Loop
Chapter 5	Claude Code, Chat, and Cowork: How Loop Engineering Helps Software Testing Engineers Get Better Results	Establish the Prompt Improvement Loop
Chapter 5	Loop Engineering for Software Testing Engineers: Building AI Systems That Improve Through Repetition	Establish the Prompt Improvement Loop
Chapter 5	Claude Code, Chat, and Cowork: How Loop Engineering Helps Software Testing Engineers Get Better Results (repeat listing on Track 2 page)	Establish the Prompt Improvement Loop

What Completion Looks Like

Track 2 is complete when the reader can demonstrate one tester-AI workflow whose prompts, context, memory, and evaluation behavior are useful, bounded, testable, measurable, and reversible. The goal is not maximum token reduction or maximum output. The goal is reliable quality engineering with controlled AI assistance.

The finished system

A real QA task enters through a defined brief; receives stable prompt policy and approved context; uses maintained project memory; follows a phase-gated workflow; exposes assumptions and unknowns; passes validation and human review; produces traceable evidence; and feeds failures back into the evaluation set.

Next improvements

- Expand the evaluation set before expanding workflow scope or autonomy.
- Add context only when existing evidence cannot support the decision reliably.
- Compare compact formats and models with the same cases before changing the production route.
- Reduce token and cost waste after quality is stable, not before.
- Promote production incidents and human corrections into permanent regression cases.
- Revisit memory, permissions, review boundaries, and rollback rules whenever the workflow changes.

Continue with the source articles and future Track 2 updates at

ValidTec.net