ABCLOUDZ

How one developer ran an end-to-end legacy migration with agentic AI

Technical case study

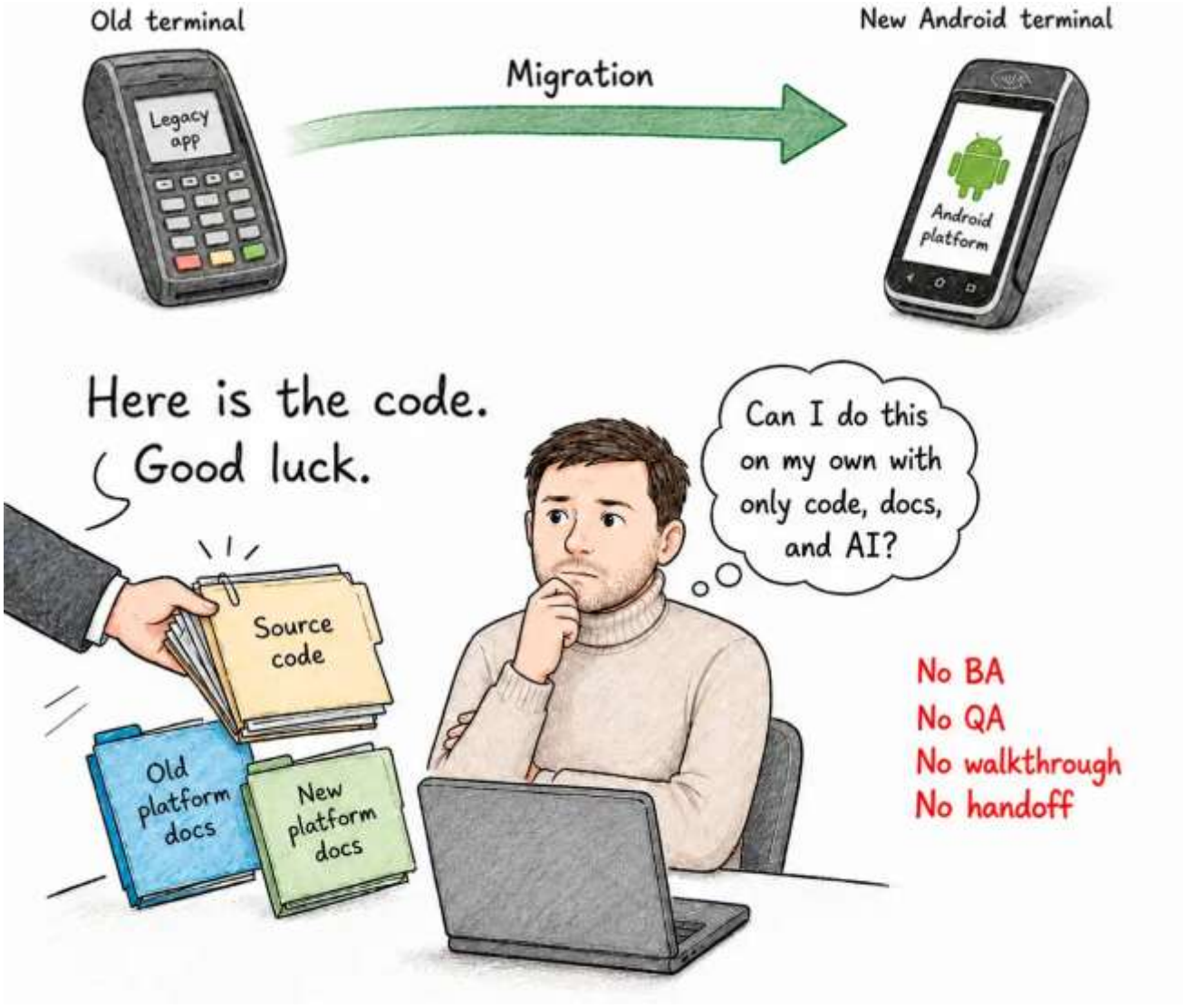
PROJECT	Migration of a legacy C++ payment application to a modern Android-based terminal platform
DELIVERY MODEL	One ABCloudz developer supported by an agentic AI workflow
AUDIENCE	Engineering leaders, architects, and product owners responsible for legacy modernization
CLIENT	Payment technology company, anonymized

Legacy modernization | Agentic AI | Android payment terminals

Contents

Executive summary

A payment technology company needed to move an established C++ payment application from an older generation of terminals to a modern Android-based platform. The initial handoff contained the source code and specifications for the old and new platforms. It did not include a product walkthrough, an authoritative requirements baseline, or a project team that could explain the system.



The assignment appeared to be a platform port. The source code revealed a broader problem. Application behavior was distributed across transaction state, device APIs, host protocols, persistent files, security mechanisms, feature flags, and recovery paths. A direct AI-generated rewrite could have produced convincing Kotlin code while preserving the wrong behavior.

I therefore separated discovery from implementation.

First, I built an evidence-backed model of the legacy application. I used general-purpose AI agents under reusable skills to analyze the code and specifications, organize findings in a linked project wiki, and run additional cross-check passes. I reviewed important interpretations against the original sources.

Next, I compared every relevant capability with the constraints of the target Android terminal platform. That analysis classified what could be rewritten, what required exact contract compatibility, what the platform already supplied, what needed an external decision, and what should be dropped.

Only after those boundaries were documented did I design the target application forward. Architecture decisions became features, requirements, flows, acceptance criteria, and ordered tasks. Each task retained links to the evidence and decisions that explained why it existed.

Implementation then ran through a controlled agentic loop. An orchestrator selected an eligible task and prepared an isolated worktree and task brief. An engineer agent implemented and self-verified the change. A separate code reviewer checked architecture, scope, defects, and tests. A QA agent verified the acceptance criteria against the running application. Any finding returned the task to the engineer, followed by a complete review and QA cycle.

This approach turned a limited handoff into:

- an evidence-backed as-built model of the legacy application;
- a linked project knowledge base;
- a capability-level portability map;
- a target architecture for the Android platform;
- an ordered implementation backlog;
- isolated, traceable task briefs;
- independent implementation, review, and QA gates;
- a repeatable workflow with explicit stop conditions and human control.

This case study documents the method, technical decisions, and implementation system. It does not claim that every production rollout activity was complete at the time of the documented project snapshot. Major foundations and transaction flows had been implemented and verified, while several security and vendor-dependent decisions remained outside the autonomous loop.

> **Central lesson**
> Understand first. Migrate second. Verify every step.

Technical environment at a glance

Area	Project choice
Legacy application	C++ payment-terminal application
Target application	Kotlin application for an Android-based payment terminal platform
Agentic development environment	Claude Code
Knowledge system	Markdown-based, two-sided linked project wiki
Target UI	Jetpack Compose
Persistence	Room or SQLite with explicit migration rules
Background and recovery work	Android lifecycle patterns and WorkManager
Dependency injection	Koin
Payment functions	Platform payment SDK and secure data interface
Verification	Gradle builds, unit tests, UI tests, mock flavor, emulator checks, layout inspection, screenshots
Source control isolation	Git worktrees per implementation task

1. The assignment was smaller than the actual problem

The request was simple:

> Build the application for the new terminals, and use AI to help.

I received the legacy C++ source code, documentation for the existing terminal platform, and documentation for the new Android-based platform. That was essentially the entire handoff.

There was no walkthrough of the application. Nobody was assigned to explain its transaction flows, host integrations, data formats, recovery behavior, security assumptions, or operational edge cases. I was working without a business analyst, QA engineer, designer, or another developer who already understood the system.

The first decision was therefore not about Kotlin, Android, or AI tooling. It was about the nature of the problem.

The apparent problem

The apparent problem was language and platform conversion:

- C++ to Kotlin;
- legacy terminal APIs to Android APIs;
- old UI technology to a modern Android UI;
- legacy storage to a new persistence layer.

The actual problem

The actual problem was missing system knowledge.

A payment application is not a collection of independent functions. Behavior can emerge from interactions between:

- transaction state;
- card-entry mechanisms;
- host authorization;
- security and cryptography;
- device lifecycle;
- offline storage;
- settlement;
- receipt generation;
- merchant configuration;
- POS or ECR integration;
- feature flags;
- restart and recovery paths.

A function could look portable in isolation and still participate incorrectly in the complete transaction flow. A generated module could compile and pass tests written around its own assumptions while failing to preserve an undocumented external contract.

That changed the project from a code conversion exercise into a reconstruction and migration program.

The first tradeoff

I had two possible starting points.

Option	Advantage	Risk
Ask AI to rewrite legacy modules immediately	Visible code appears quickly	There is no verified baseline for judging correctness
Reconstruct the existing system before generating target code	Creates requirements, boundaries, and evidence	Delays visible implementation work

The first option optimized for output. The second optimized for correctness.

I chose the second.

2. I separated discovery from implementation

The most tempting AI-first move was to feed legacy modules into the model and ask for Kotlin replacements.

I rejected that path because it would have created two unknown systems at once:

1. a legacy application I did not yet understand;
2. a generated Android application whose behavior I could not reliably evaluate.

The source code was the strongest available evidence, but source code is not a requirements document. It can show what happens. It rarely explains why the behavior exists, which parts are contractual, or which parts are historical artifacts of the old platform.

I introduced a strict boundary between discovery and implementation.

Discovery had to establish a baseline

Before a feature could move into implementation, I needed to know:

- its business or operational purpose;
- its boundaries and dependencies;
- the relevant legacy evidence;
- the target-platform constraints;
- applicable architecture decisions;
- observable acceptance criteria;
- unresolved questions that still required a decision.

The practical test was simple:

> Could an engineer implement the feature, and could another person independently determine whether the result was correct?

If either answer was unclear, the feature remained in discovery.

Why this mattered for AI

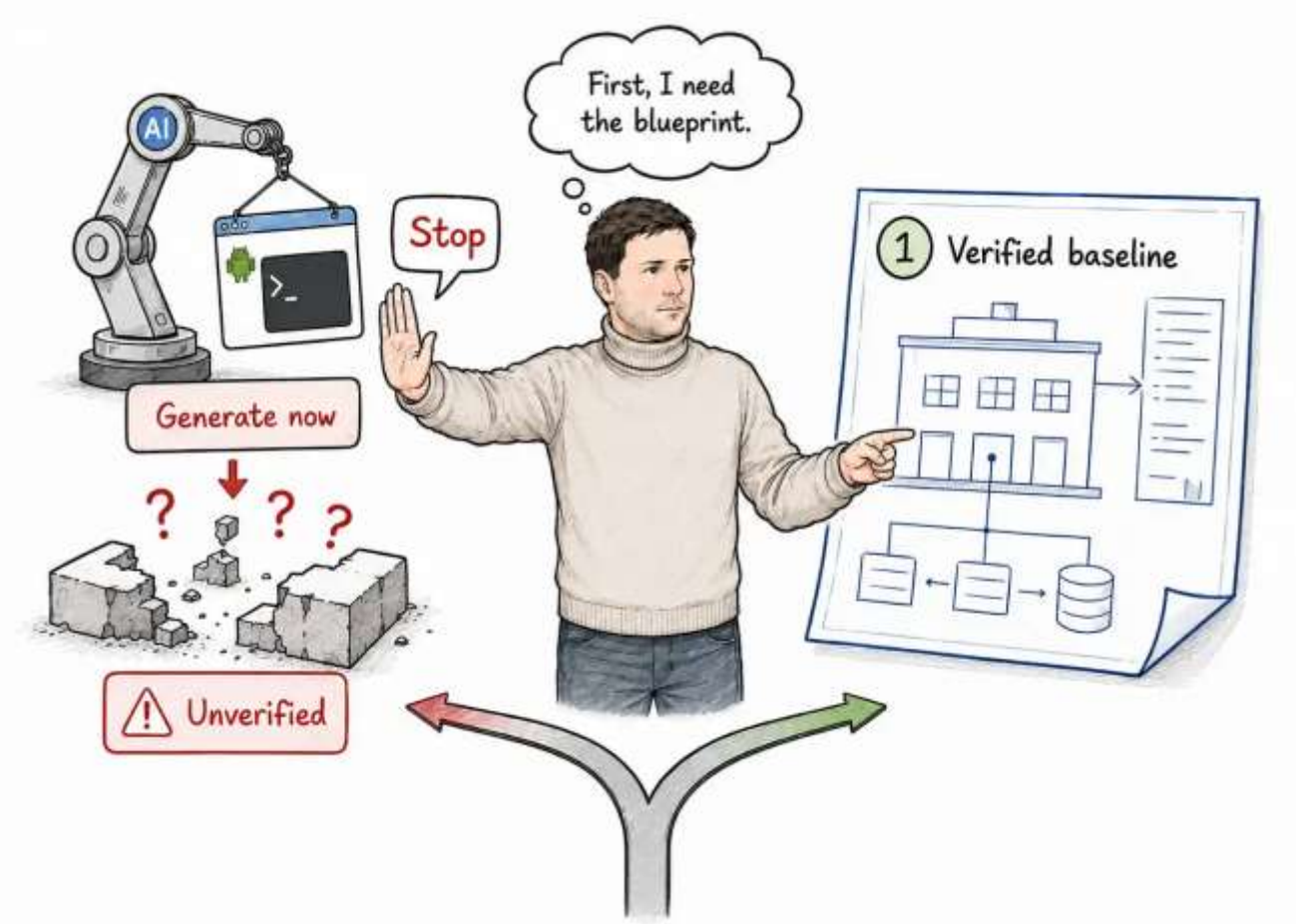
AI is effective when the work has explicit boundaries and evidence. It becomes risky when ambiguity is silently converted into implementation assumptions.

Separating discovery from implementation changed the AI role.

During discovery, AI acted as an analyst and knowledge organizer. It extracted behaviors, followed dependencies, compared documentation, and drafted structured findings.

During implementation, AI acted against bounded tasks, architecture rules, and acceptance criteria.

This distinction prevented generated code from becoming the place where requirements were accidentally invented.



Decision record

Decision: Build a verified baseline before code generation.
Rejected alternative: Translate the codebase module by module.
Reason: A plausible rewrite could preserve syntax and structure while changing system behavior.
Result: Implementation started from documented features and acceptance criteria rather than inferred behavior inside generated code.

3. I built an AI operating system for the project

The next question was practical:

> How could one developer analyze a large codebase and two platform documentation sets without repeatedly loading the entire project into every AI context?

A single large prompt was not a durable answer. It would be difficult to review, expensive to reload, and vulnerable to context loss.

I needed two foundations:

1. reusable skills that defined how agents should work;
2. a linked project wiki that preserved what the project had learned.

Skills defined behavior

The system did not depend on long-lived named agents with hidden memory. I used general-purpose agents and defined each working role through:

- the model selected for the task;

- a reusable skill;
- a focused prompt or task brief;
- explicit acceptance criteria;
- a structured report.

This made agent behavior reproducible. A new agent could enter the project and follow the same procedure without relying on a previous conversation.

The three core wiki skills

wiki-management

This skill controlled the lifecycle and structure of project documentation.

It defined:

- document types and templates;
- file naming and identifiers;
- metadata headers;
- side-specific tags;
- directory indexes;
- links between related artifacts;
- rules for creating, updating, moving, and deleting pages.

The wiki was divided into two sides:

- the legacy side, containing as-built reference material;
- the target side, containing forward-looking architecture and delivery documentation.

The separation was important. It prevented descriptions of current behavior from being confused with decisions about future behavior.

wiki-reader

The wiki became too large to load indiscriminately. The reader therefore followed an index-first retrieval process.

It first identified whether the question concerned:

- the legacy C++ application;
- the new Android application;
- both sides;
- project tasks and status.

It then opened the appropriate index, selected the smallest relevant set of pages, and followed only the links required for the current question.

This was not only a token-saving mechanism. It reduced irrelevant signals and made the context easier to audit.

wiki-validator

The validator had a narrower role.

It checked structural integrity, including:

- required metadata;
- valid identifiers and naming;
- directory placement;
- side tags;
- index membership;
- internal links;
- reference format.

It did not decide whether a technical conclusion was true. Semantic validation remained the responsibility of analysis passes and human review.

Supporting documentation skills

Separate reader skills handled the external platform corpora:

- one for the Android terminal platform documentation;
- one for the payment SDK and secure data interface documentation.

They used the same retrieval principle: index first, smallest relevant document set second.

The linked wiki as project memory

The wiki was not a transcript archive. It was a structured engineering model.

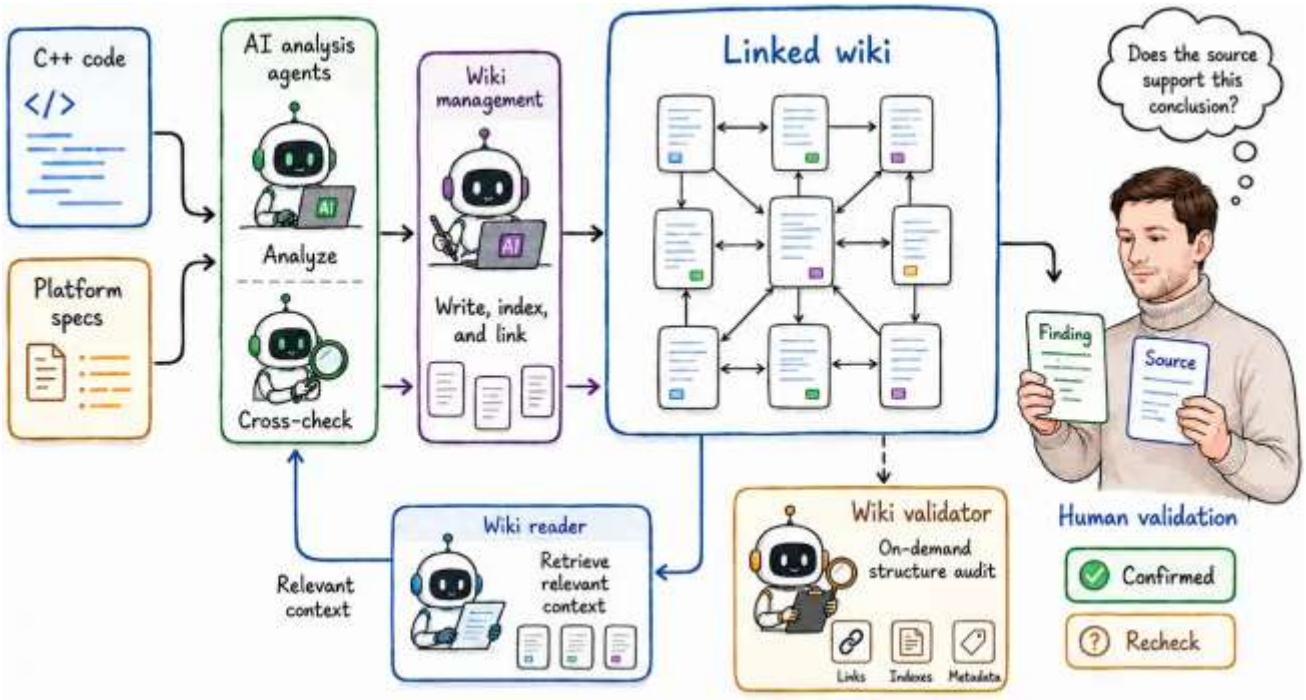
The legacy side included artifacts such as:

- architecture;
- data model;
- technologies;
- glossary;
- modules;
- APIs;
- behaviors;
- functional and nonfunctional requirements;
- features;
- legacy architecture decisions.

The target side included:

- target architecture;
- data model;
- technologies;
- product design;
- requirements;
- user stories;
- user flows;
- features;
- wireframes;
- architecture decisions;
- roadmap and execution plan.

The task area connected the design to implementation work.



The developer's role during discovery

AI performed much of the extraction and organization, but I remained responsible for curation.

My work included:

- deciding which questions mattered;
- reviewing ambiguous interpretations;
- comparing findings with code and specifications;
- rejecting weak or unsupported conclusions;
- deciding when another analysis pass was required;
- ratifying architecture decisions;
- controlling priorities.

This phase contained substantial human involvement because errors in the knowledge base would propagate into every later task.

4. I reconstructed the legacy application backward

Normal product development moves forward:

> requirements → architecture → implementation

I had to move in the opposite direction:

> implementation → behavior → requirements → architecture

The code and platform specifications became evidence from which the system had to be reconstructed.

The analysis sequence

The analysis did not happen in one pass.

Pass 1: inventory

Agents identified major modules, entry points, data structures, external interfaces, device integrations, and persistent artifacts.

The goal was not yet to explain the whole system. It was to establish a map of where important responsibilities appeared to live.

Pass 2: behavior reconstruction

Agents followed call paths and shared state to recover runtime behavior.

This included questions such as:

- How does a transaction start?
- Which state transitions are possible?
- When is host authorization invoked?
- Where are reversals or advice messages triggered?
- How is offline work retained?
- What happens after restart or interruption?
- Which values are persisted across sessions?

Pass 3: interface and contract analysis

The next pass focused on boundaries:

- host wire protocol;
- POS or ECR framing;
- device APIs;
- card-entry interfaces;
- storage layouts;
- configuration formats;
- cryptographic responsibilities.

This distinction mattered because an internal implementation could be redesigned, while an external contract might need exact compatibility.

Pass 4: feature and requirement reconstruction

The behavior findings were organized into features, functional requirements, nonfunctional requirements, user flows, and architecture decisions.

The resulting documents linked back to the code and specification evidence.

Pass 5: cross-check

Additional agents reviewed the earlier analysis for omissions and unsupported conclusions.

They looked for:

- modules that had not been mapped;
- flags or structures with unexplained consumers;
- behaviors that appeared in code but not in documentation;
- contradictory interpretations;
- references that did not resolve;
- system effects hidden behind local names.

How one conclusion was validated

An opaque flag might initially appear to control one local setting.

That interpretation was only a hypothesis.

I would trace:

1. where the value was assigned;
2. where it was read;

3. which modules depended on it;
4. which transaction states it affected;
5. whether related behavior appeared in the platform specifications.

The broader reference path could reveal that the same value also affected initialization, recovery, or host communication.

Only after that investigation could the finding become part of the as-built model.

The validation question remained:

> Can this conclusion be followed from the wiki back to observable evidence?

When the answer was unclear, the finding returned for another analysis pass.

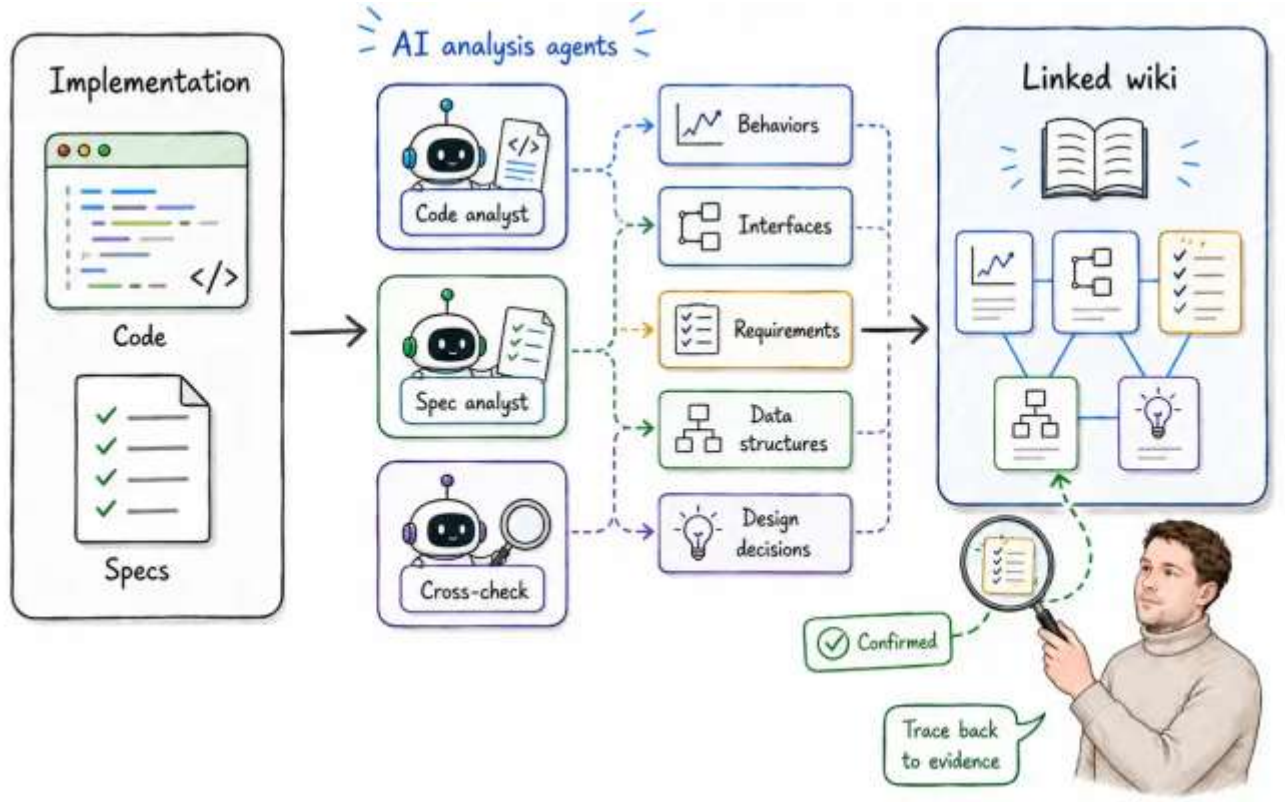
Why traceability mattered

A migration requirement without evidence is difficult to trust.

Traceability provided three benefits:

- reviewers could inspect the basis of a conclusion;
- target decisions could link to the legacy behavior they replaced;
- implementation tasks could explain why they existed.

This turned the wiki from a documentation repository into an engineering control system.



What the as-built model contained

The reconstructed system model included, among other areas:

- transaction lifecycle and type catalog;
- card acceptance and cardholder verification;
- host authorization;
- offline store-and-forward;
- settlement and batch management;
- receipts and reporting;

- POS or ECR integration;
- multi-merchant configuration;
- terminal initialization;
- device security and recovery;
- key loading and cryptography;
- data structures and persistent formats.

The as-built model became the foundation for migration analysis.

5. I classified portability instead of treating migration as translation

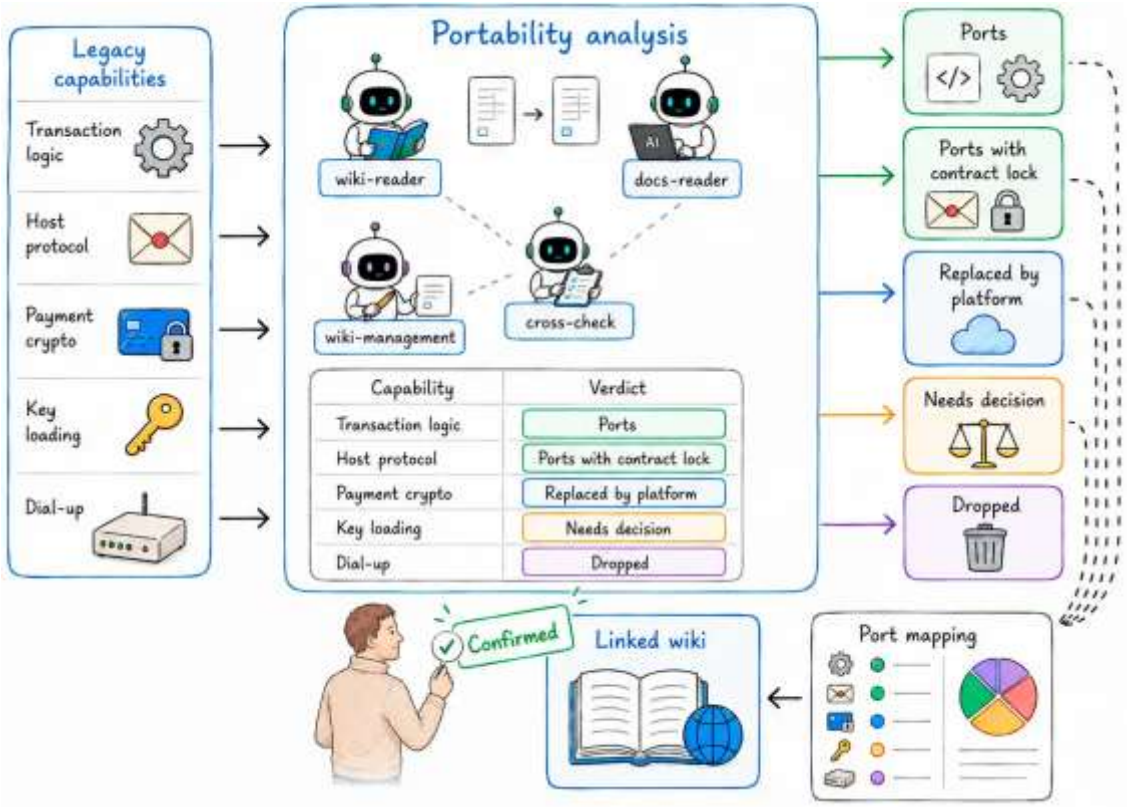
Once the legacy application was understood, the next question was not:

> How do I rewrite this in Kotlin?

The better question was:

> What is the correct target treatment for each capability?

The new platform imposed constraints that did not exist in the legacy environment.



Examples included:

- payment-sensitive operations isolated behind a secure processor;
- card readers, PIN entry, key storage, and payment crypto exposed through secure APIs;
- no application-supplied EMV kernel;
- no direct device-node access for payment hardware;
- privileged application and signing requirements;
- no legacy modem or dial-up path;
- Android lifecycle, Doze, restart, and resource constraints;
- exclusive access rules for secure payment functions.

A uniform rewrite would have ignored those boundaries.

The five verdicts

I used five primary portability verdicts.

Verdict	Meaning
Ports	The logic survives, but the platform binding is rewritten using Android or target-platform APIs
Ports with contract lock	The capability can move only if a byte-level wire or data contract remains exact
Replaced by platform	The target platform now owns the capability, so the application calls the platform service instead of reimplementing it
Needs decision	Progress depends on a vendor, security, or architecture choice
Dropped	The capability is retired, unsupported, or prohibited on the target platform

The verdict was applied at capability level, not file level.

Representative mappings

Legacy capability	Portability verdict	Target treatment
Transaction lifecycle and transaction types	Ports	Kotlin domain model and typed state machine
Host authorization over TLS	Ports	Android networking client
Proprietary host message format	Ports with contract lock	Custom codec that reproduces the existing contract
POS or ECR framed protocol	Ports with contract lock	Android Bluetooth or socket transport with the same frame rules
Card reading and EMV processing	Replaced by platform	Payment SDK and secure data interface
PIN entry and payment-data encryption	Replaced by platform	Secure processor APIs
Key loading	Needs decision	Vendor and sponsor-controlled key process
Legacy dial-up	Dropped	Modern network transports only
Legacy UI	Ports	Jetpack Compose
Flat-file persistence	Ports or contract-sensitive migration	Room or SQLite, with compatibility rules where required

The largest perceived blocker changed category

The legacy documentation treated the external EMV kernel as a major migration decision. A direct porting mindset suggested that the target application might need an equivalent kernel integration.

The target platform documentation changed the answer.

The Android terminal platform already supplied certified EMV L2 processing through its payment SDK and secure data interface. The application did not bring its own kernel.

That meant the following legacy components did not port:

- the external kernel transport;
- the legacy packet driver;
- the kernel queueing mechanism.

What remained in the application was the result handling:

- interpretation of returned data;
- cardholder verification outcome;
- cryptogram and transaction result routing;
- host request construction.

A perceived application blocker became a platform integration decision.

The EMV kernel and payment host were different problems

The EMV kernel and the payment host participate in the same transaction, but they perform different jobs.

EMV kernel

The kernel conducts the local conversation with the payment card. It handles activities such as application selection, risk processing, cardholder verification, and cryptogram generation.

On the target platform, this capability was replaced by the platform.

Payment host

The payment host authorizes the transaction and routes it to the relevant payment infrastructure.

That external system remained. Its communication contract still had to be preserved.

The resulting decisions were different:

- EMV processing was **replaced by platform**.
- The payment-host client **ported with contract lock**.

Confusing those layers would have produced the wrong risk model.

6. I designed the target application forward

After reconstructing the legacy application backward, I could return to the normal direction of software design.

The migration analysis now provided two essential answers:

- what behavior needed to survive;
- how the target platform allowed that behavior to be implemented.

From there, I designed the new application forward:

> architecture → features → requirements → flows → acceptance criteria → tasks

Target architecture decisions

The target design replaced legacy platform patterns with Android-native or platform-controlled components.

Representative decisions included:

- Kotlin domain objects instead of global C structs;
- a typed state machine instead of table-driven dispatch structures;
- Jetpack Compose for the UI;
- Room or SQLite for durable application data;
- WorkManager and explicit recovery behavior for background or resumed work;
- dependency injection for platform-specific wiring;
- a platform API boundary separating application logic from vendor SDK types;
- separate real-device and mock platform implementations;
- payment SDK and secure processor APIs for card, PIN, and sensitive cryptographic work.

The key principle was not to imitate the old architecture. It was to preserve required behavior and contracts while using an architecture appropriate for the new platform.

The two-sided traceability model

A target feature linked backward to the legacy evidence and sideways to the target design.

A security-related feature, for example, could link to:

- the legacy security feature;
- the legacy host protocol;
- the architecture decision for platform-managed EMV;
- related target features;
- user stories;
- a tamper recovery flow;
- sensitive-data requirements;
- a PIN entry wireframe;
- decisions about host payload crypto and key loading.

This linking model prevented architecture decisions from becoming detached from the behavior and constraints that created them.

Why architecture decisions were separate documents

A feature document should describe what the system needs to do.

An architecture decision record should explain:

- the decision;
- the alternatives considered;
- the constraints;
- the reason one option was selected;
- the consequences.

Keeping those concerns separate made it possible to revise a decision without rewriting the feature's purpose.

7. I turned the knowledge base into an executable plan

A complete knowledge base was still not an implementation plan.

The next step was to convert architecture and features into ordered, bounded work.

From evidence to task

The traceability chain looked like this:

> legacy evidence → portability verdict → architecture decision → target feature → requirements and flow → acceptance criteria → task brief

This chain gave each task a reason to exist.

A task was not merely a line in a backlog. It was connected to:

- the behavior being preserved or replaced;
- the target architecture;
- relevant constraints;
- acceptance criteria;
- source documents.

The execution tracker

The execution plan tracked each task with fields such as:

- task ID;
- task name;

- status;
- blocking reason;
- complexity;
- assigned model;
- parallel eligibility.

Status values distinguished:

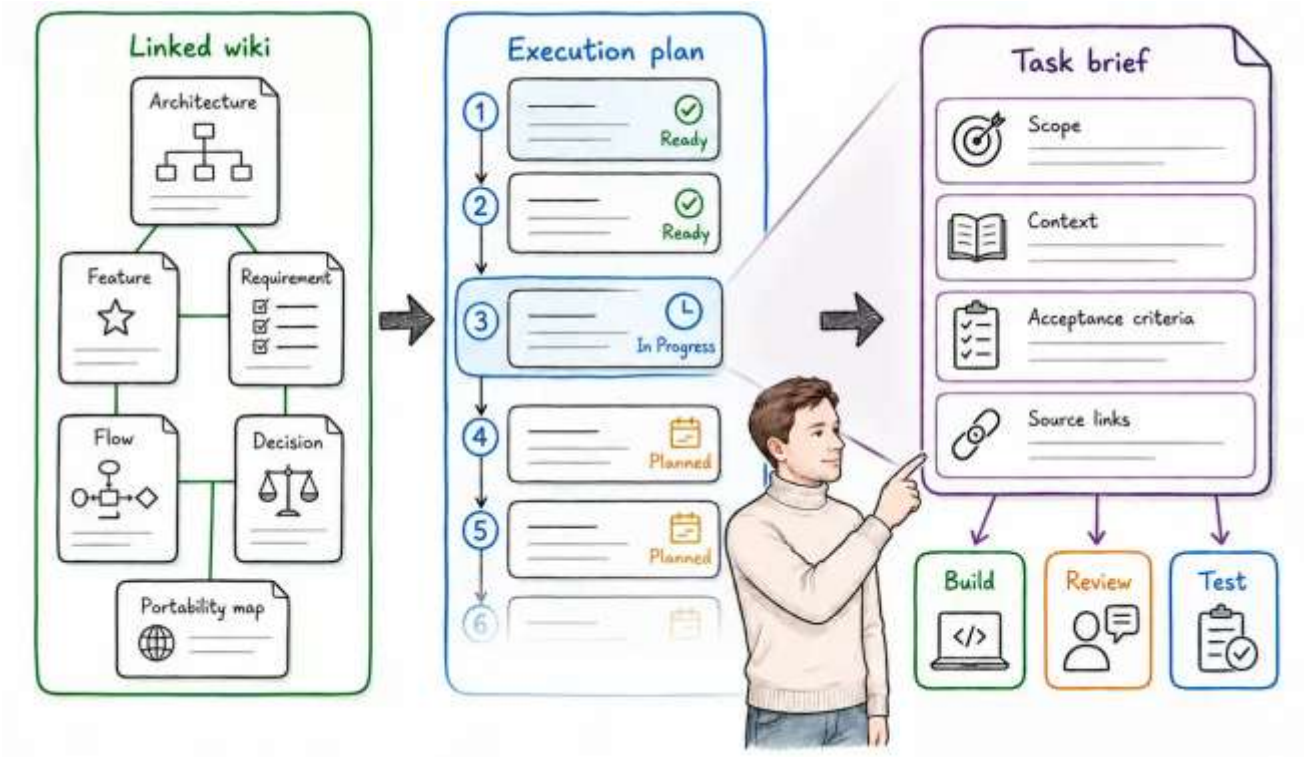
- backlog;
- done;
- blocked;
- deferred.

Complexity and model selection were planning inputs, not cosmetic labels. More critical work could be assigned to a stronger model. QA used the stronger model by default.

Parallel eligibility was re-evaluated as dependencies changed. However, the configured implementation loop intentionally took one selected task fully to green before starting the next. This kept repository state, reports, and task status synchronized.

The task brief

Before an implementation agent started, the orchestrator created an isolated worktree and wrote `.task-brief.md`.



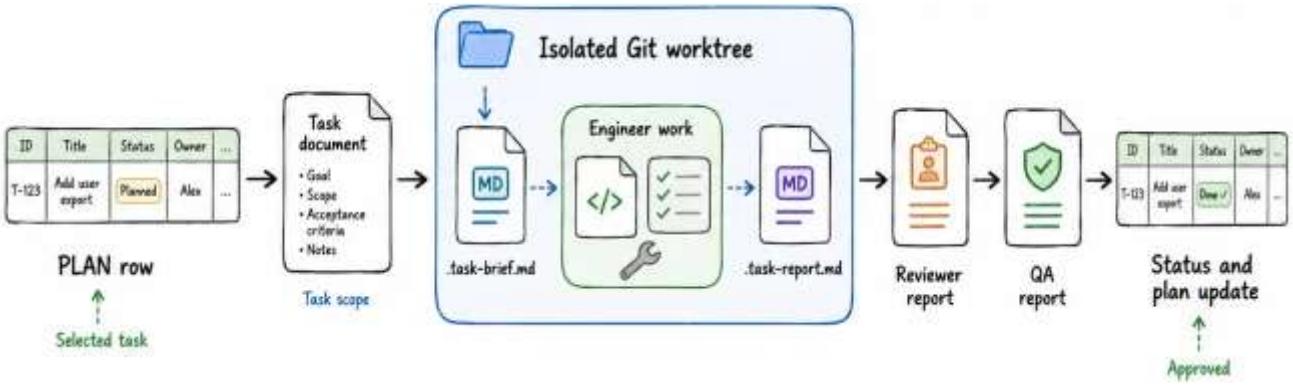
The brief contained:

- task ID and title;
- implementation objective;
- scope or a pointer to the authoritative wiki page;
- acceptance criteria;
- constraints;
- implementation hints when required.

The prompt referenced the file path rather than copying a long brief into the orchestrator context.

For a documented task, the agent was expected to retrieve the relevant wiki pages before planning the implementation.

This kept prompts short and made the wiki, not the prompt, the source of truth.



Why isolated worktrees mattered

Every implementation agent worked in its own Git worktree.

This provided:

- isolation from the main working tree;
- a clean baseline;
- a stable location for brief and report files;
- safer review before changes were applied;
- easier cleanup after completion.

The dispatcher also copied required untracked build files into the worktree, such as local Android configuration.

8. I organized implementation as independent roles

The implementation system used four functional roles:

1. orchestrator;
2. engineer;
3. code reviewer;
4. QA engineer.

These were not persistent custom agent identities. They were general-purpose agents differentiated by model, skill, task brief, acceptance criteria, and report format.

Orchestrator

The orchestrator used the `implementation-loop` and `agent-dispatch` procedures.

Its responsibilities included:

- selecting the next eligible task;
- checking status and dependencies;
- choosing the model defined in the plan;
- preparing the worktree;
- writing the task brief;
- dispatching the implementation role;
- reading the full report files;

- dispatching review and QA;
- returning findings to the engineer;
- updating project status after all gates passed;
- stopping when a human decision or external blocker was required.

The orchestrator did not write the feature code itself.

Engineer

The engineer used `task-implementation`.

Its required sequence was:

1. analyze the brief, wiki, and source files;
2. clarify missing information before coding;
3. verify that the task was doable;
4. write a concise implementation plan;
5. implement only the defined scope;
6. run build and tests;
7. verify UI work on a running emulator or device;
8. self-review against acceptance criteria;
9. write `.task-report.md`.

The report recorded:

- completion status;
- summary;
- files changed;
- verification performed;
- concerns;
- blockers.

The engineer could return one of four states:

- `DONE`;
- `DONE_WITH_CONCERNS`;
- `NEEDS_CONTEXT`;
- `BLOCKED`.

Code reviewer

The code reviewer worked independently in the same worktree.

It checked:

- conformance with architecture and ADRs;
- potential defects;
- missing scope;
- scope creep;
- test coverage and meaning;
- build status;
- regression gates;
- repository standards.

The reviewer did not rely on the engineer’s self-assessment.

QA engineer

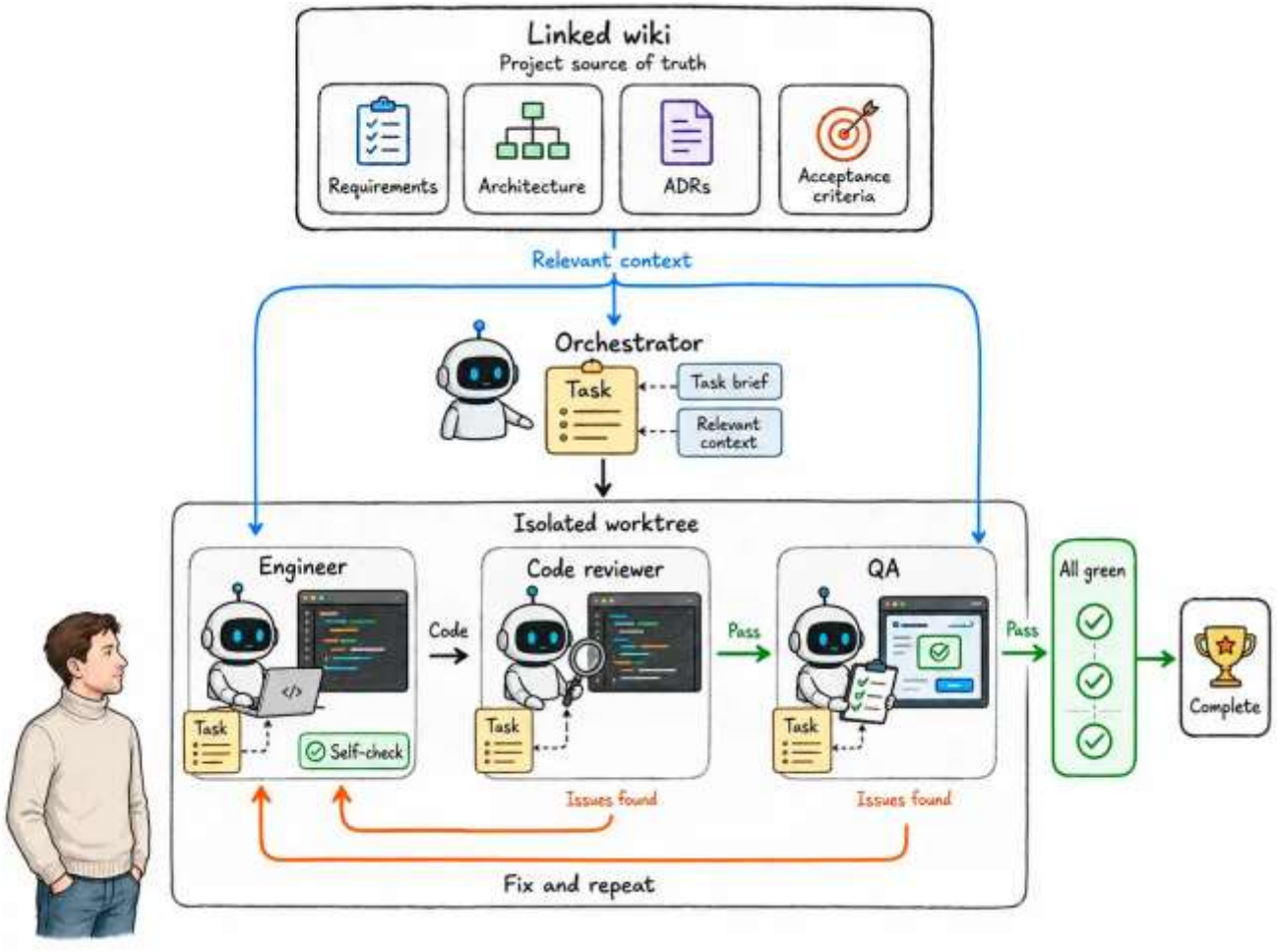
QA verified the feature against the acceptance criteria and the running application.

For UI work, the project used a mock product flavor that could run on a standard Android emulator without the vendor payment SDK.

QA could:

- launch the app;
- drive flows;
- inspect the UI layout tree;
- capture screenshots;
- run UI tests;
- execute deterministic mock scenarios;
- confirm that existing behavior remained intact.

The layout tree was treated as the primary deterministic source for visible text, identifiers, and bounds. Screenshots confirmed appearance.



The fix cycle

If code review or QA found an issue, the task returned to the engineer in the same worktree.

After the fix:

1. the engineer self-verified again;
2. code review ran again;
3. QA ran again.

A task could not jump directly from a fix to completion.

This mattered because a fix could resolve one problem while introducing another.

All-green completion

A task was complete only when:

- engineer self-verification passed;
- code review passed;
- QA passed.

The process then:

- copied approved changes from the worktree;
- synchronized task and plan status;
- recorded progress;
- created a local commit under the project's explicit approval rules;
- removed the worktree;
- did not push.

Standing technical gates

The implementation loop included project-wide gates in addition to task-specific acceptance criteria.

Examples included:

- dependency-injection graph resolution for both real and mock flavors;
- confirmation that the mock build remained free of vendor SDK dependencies;
- protection-by-default rules preventing clear cardholder data, track data, CVV, or PIN from crossing the platform boundary or entering application state.

These gates made security and architecture constraints part of every task, not separate cleanup work.

Stop conditions

Autonomy had explicit boundaries.

The loop stopped when:

- the context window exceeded the configured threshold;
- repeated fix cycles were not converging;
- an architectural conflict could not be resolved safely;
- an external dependency blocked progress;
- a product, vendor, or security decision required human judgment.

The purpose of a stop condition was not to reduce automation. It was to stop the system from turning uncertainty into confident but unsafe work.

9. The human role became narrower and more important

The project was executed by one developer, but that did not mean the developer manually performed every engineering activity.

The AI agents supplied analysis, drafting, implementation, independent review, and QA capacity.

The human supplied:

- project direction;
- interpretation of ambiguous evidence;
- curation of the source of truth;

- architecture judgment;
- prioritization;
- decision ownership;
- external coordination;
- acceptance of risk.

This division was essential.

What I delegated

I delegated work that could be bounded and verified:

- code and documentation analysis;
- structured document creation;
- cross-check passes;
- implementation against acceptance criteria;
- code review;
- test execution;
- UI verification;
- report generation.

What I retained

I retained work that depended on context, accountability, or product judgment:

- deciding what questions needed answers;
- determining whether evidence was sufficient;
- resolving conflicting interpretations;
- ratifying ADRs;
- deciding how to handle external constraints;
- approving changes under project rules;
- deciding when to stop or re-scope.

Why one source of truth mattered

With many agents operating on the project, the main coordination risk was contradictory context.

The linked wiki and execution tracker reduced that risk by giving every role the same authoritative reference.

The human did not have to repeat the project in every prompt. The human had to maintain the quality and direction of the source material that prompts referenced.

10. What the approach produced

The project began with:

- legacy C++ code;
- two platform documentation sets;
- one developer;
- no transferred product knowledge;
- an unknown migration scope.

The process produced a documented and executable migration program.

Knowledge outputs

- legacy architecture;
- feature and behavior inventory;
- interfaces and external contracts;
- data model;
- functional and nonfunctional requirements;
- legacy decisions and constraints;
- traceable source links.

Migration outputs

- capability-level portability verdicts;
- target-platform constraints;
- contract-lock requirements;
- platform replacement decisions;
- vendor and security decision points;
- dropped capabilities.

Target design outputs

- target architecture;
- Kotlin domain and state-machine design;
- Android UI approach;
- persistence strategy;
- platform abstraction boundaries;
- mock and real-device implementations;
- ADRs;
- requirements;
- flows;
- wireframes;
- acceptance criteria.

Delivery outputs

- ordered backlog;
- dependency-aware execution tracker;
- model and complexity assignments;
- task briefs;
- isolated worktrees;
- engineer reports;
- review reports;
- QA reports;
- fix cycles;
- explicit stop conditions.

Implementation status at the documented snapshot

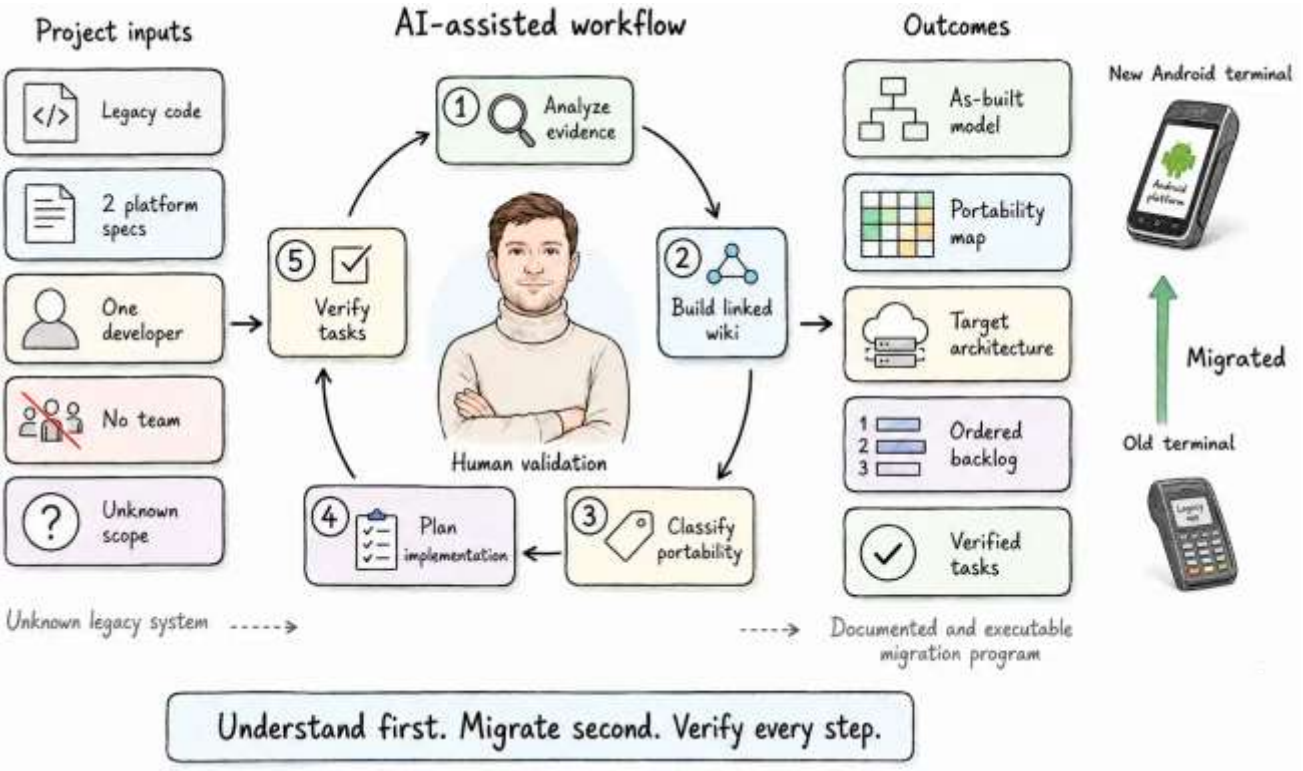
At the point represented by the internal project snapshot, the implemented and verified work included substantial parts of:

- Android multi-module foundations;
- dependency injection and product flavors;

- core primitives and sensitive-data masking;
- Room persistence and migration framework;
- transaction state and transition model;
- coroutine execution and cancellation;
- durable restart behavior;
- terminal initialization and configuration;
- card-capture abstractions;
- payment SDK integration;
- mock card-present scenarios;
- cardholder verification flows;
- transaction handlers;
- host TLS communication;
- proprietary host message codec;
- retry and duplicate-detection behavior;
- authorization result flows.

Several security and key-management tasks remained dependent on external vendor or security decisions. Later feature areas such as settlement, receipts, offline replay, device integrity, POS integration, and multi-merchant behavior remained part of the continuing migration program.

The important result was not a claim that AI had independently completed an entire production migration. The result was a controlled system that could turn documented migration work into verified implementation while keeping unresolved decisions visible.



11. The decisions that shaped the project

The project can be summarized as a series of deliberate tradeoffs.

Decision point	Rejected path	Selected path	Reason
When to generate code	Start with module conversion	Establish a verified baseline first	Generated code needed requirements and acceptance criteria
How to store project knowledge	Keep knowledge in prompts and chat history	Build a linked two-sided wiki	Durable, searchable, traceable context
How much context to load	Load the entire wiki	Retrieve the smallest relevant page set	Lower noise and easier review
How to validate documentation	Trust the first analysis pass	Run cross-check passes and human review	Legacy behavior was distributed and ambiguous
How to think about portability	Rewrite every feature	Classify each capability	Target constraints changed the correct treatment
How to handle EMV	Port the legacy kernel integration	Use platform-provided EMV	The target platform already owned certified processing
How to preserve host behavior	Replace with a generic protocol client	Reproduce the existing contract	External compatibility remained mandatory
How to define agent roles	Depend on persistent agent identities	Use model + skill + brief + criteria + report	Reproducible behavior across sessions
How to execute tasks	One agent builds and declares completion	Engineer + independent review + QA	Separate implementation from evaluation
How to handle fixes	Accept after the engineer fixes	Repeat review and QA	Fixes can introduce regressions
How to handle uncertainty	Let the model choose silently	Stop and escalate	Product, security, and vendor decisions require ownership
How to sequence implementation	Maximize concurrent code changes	Run each selected task to all green	Deterministic repository and status control

12. Lessons for other legacy modernization projects

Understanding is an engineering deliverable

When requirements are missing, reconstructing the current system is not preparation for the project. It is part of the project.

The quality of every generated artifact depends on the quality of that baseline.

Agent names matter less than operating rules

The useful unit was not a colorful agent persona. It was a reproducible role defined by:

- a goal;
- a skill;
- relevant context;
- acceptance criteria;
- a report;
- a gate.

Context architecture affects output quality

A large knowledge base is useful only if agents can retrieve the relevant part.

Index-first reading and linked documents reduced both context waste and accidental omission.

Portability is a classification problem

A migration contains different kinds of work:

- rewrite;
- exact compatibility;
- platform replacement;
- external decision;
- retirement.

Treating them as one code-conversion category hides risk.

Independent verification is essential

AI-generated code should not be evaluated only by the agent that wrote it.

Implementation, code review, and functional QA had separate mandates and reports.

Human control belongs at ambiguity boundaries

Human time was most valuable where evidence, architecture, risk, or external dependencies required judgment.

The goal was not to remove the developer. It was to move the developer from repetitive execution into curation and decision ownership.

13. When this approach is a good fit

This operating model is particularly useful when:

- source code exists but documentation is incomplete;
- a working legacy application must move to a new platform;
- the target platform changes security or hardware responsibilities;
- external protocols must remain compatible;
- one or a few experts need to coordinate more work than they can perform manually;
- the domain requires traceability and independent verification;
- migration decisions must remain linked to evidence.

It is less suitable when:

- source code and executable behavior are unavailable;
- nobody owns product or architecture decisions;
- external platform access is insufficient;
- acceptance criteria cannot be defined;
- the organization expects AI to resolve contractual or vendor decisions without accountable human input.

14. Conclusion

I began with source code, two platform specifications, and almost no transferred knowledge.

The useful breakthrough was not asking AI to write more code. It was building a process in which AI could first reconstruct the system, preserve that knowledge, classify migration decisions, convert them into bounded work, and verify each implementation step independently.

That process gave one developer the analytical and execution capacity normally distributed across several roles, while keeping evidence, architecture, and final judgment under human control.

Organizations facing a legacy migration, undocumented application, or complex platform transition can apply the same principles without copying this exact toolchain.

At ABCloudz, I work with engineers who combine modernization experience with practical agentic AI delivery. We can help structure discovery, build a reliable project knowledge base, define agent skills and verification roles, classify migration boundaries, and turn the result into an executable development process.

Bring us the code, the platform constraints, and the outcome you need. We will help turn them into a development process that is structured, traceable, and ready to execute.