The InclinePOS Failure: What Ignoring Offline Requirements Taught Me

Muhammad Gulzaib
https://orcid.org/0009-0000-8709-6989
SynergyFlow Labs · Islamabad, Pakistan

Published	2026-08-18
URL	https://gulzaib.vercel.app/blog/the-inclinepos-failure-what-ignoring-offline-requirements-taught-me
Licence	© 2026 Muhammad Gulzaib. All rights reserved. Code samples MIT.

ABSTRACT

How a cloud-first point-of-sale product missed an offline requirement, and what I would change about discovery, transactions and synchronisation.

Keywords: founder, product, lessons, saas, offline

> **TL;DR**
>
> InclinePOS failed because I treated continuous internet connectivity as a safe default for a point-of-sale system. For the businesses I wanted to serve, selling could not stop when the connection did. The lesson was to test operational requirements before building the architecture around them.

This article was prepared in November 2026. The displayed 18 August 2026 date was requested for the post; it is not its original publication date.

InclinePOS did not fail because the code was fundamentally broken. It failed because I built around a technically reasonable assumption that did not match how customers actually operated.

That distinction changed how I think about product development. A product can have a clean architecture, a modern interface, reliable APIs, and well-structured code, yet still be unusable in the environment it was designed to serve.

The product made sense on paper

InclinePOS was a cloud-based point-of-sale system intended to help businesses process sales, manage products, track inventory, record transactions, monitor performance, and centralise operational data.

From a software perspective, the cloud-first model was attractive. A central database

simplified synchronisation. Updates could be deployed without visiting every customer. Business owners could access their data from different devices. Backups, reporting, permissions, and future integrations were easier to manage.

The architecture was rational. The problem was treating continuous connectivity as a safe default.

The assumption hidden inside the architecture

InclinePOS contained a major assumption: the customer would have a sufficiently stable internet connection whenever the system was needed.

That assumption influenced how transactions were created, where data was validated, when inventory was updated, how authentication worked, and what happened when the application could not reach the server. As long as connectivity existed, the product could perform as intended.

A point-of-sale system sits directly in the critical path of revenue. If a reporting dashboard becomes temporarily unavailable, a manager may wait. If a POS system becomes unavailable while customers are standing at the counter, the business cannot wait.

The shop still needs to sell. The cashier still needs to create a receipt. Inventory still needs to be adjusted. At the moment of sale, the POS becomes part of the business's operational infrastructure. Availability is part of the product's core value proposition.

What "offline support" actually means

It is tempting to treat offline functionality as a small feature: cache a few screens, save some data locally, show an offline indicator, and synchronise when the connection returns. For a transactional system, offline support is an architectural commitment.

Where are transactions stored?

If the server is unreachable, a transaction must be stored on the device. The local data must survive browser refreshes, application restarts, device sleep, temporary crashes, and failed synchronisation attempts.

Surviving a restart is not the same as surviving a lost or destroyed device. If the only copy of an unsynchronised sale is on that device, recovery needs a separate backup or replication plan.

How are transaction identifiers generated?

A system cannot rely only on the server to assign sequential IDs when multiple devices may create transactions independently while offline. Possible approaches include client-generated UUIDs, device-specific identifiers, temporary local IDs mapped to server IDs, and

idempotency keys to prevent duplicate submissions.

What happens to inventory?

Suppose two counters sell the last unit of the same product while both devices are offline. Each device may believe the item is available. When the connection returns, the server discovers that the combined sales pushed inventory below zero.

The system needs explicit rules: allow temporary negative inventory, flag the conflict for review, reserve stock per terminal, use a central local-network server, or prevent certain operations while disconnected. There is no universal answer. The right decision depends on how the business operates.

How are prices and discounts handled?

A cashier may apply a discount while offline. Meanwhile, an administrator could change the product price from another location. When the records synchronise, which version should win?

The system needs rules for price snapshots, discount permissions, tax calculations, promotion validity, manager approvals, and audit trails.

What if synchronisation fails halfway?

Uploading a transaction is not enough. The application must know whether the server successfully processed it. Without idempotent operations, a retry could create duplicate sales. Without an audit trail, partial failures may be difficult to reconstruct. Without visible synchronisation status, the cashier may assume the data is safe when it is still stored only on one device.

Offline functionality affects the data model, interface, security model, conflict rules, testing strategy, and deployment architecture. By the time I treated it as important, it required reconsidering foundational decisions.

Technical correctness is not product correctness

InclinePOS forced me to separate two ideas I had previously treated as closely related.

A technically correct product may have well-structured code, secure authentication, accurate calculations, a normalised database, good API design, automated deployments, responsive interfaces, and reliable cloud infrastructure.

A product-correct system fits the customer's environment. It works during their operating hours, survives the failures they routinely experience, fits their workflows, respects the hardware they own, and handles the conditions under which the business makes money.

InclinePOS may have contained technically sound decisions, but it was not sufficiently

aligned with the conditions in which customers needed to use it. The market does not reward architectural elegance in isolation. It rewards systems that remain useful when reality becomes inconvenient.

The cost of discovering a requirement too late

Some requirements can be added after launch: a new report, a different receipt layout, another export format, or a dashboard filter. Others determine the shape of the system: offline operation, multi-tenancy, data residency, auditability, hardware compatibility, or regulatory compliance.

Discovering an architectural requirement late creates a multiplier effect. For InclinePOS, real offline capability would affect:

1. **Storage:** a durable local database rather than temporary interface state.
2. **Transaction processing:** sales created locally and submitted safely later.
3. **Inventory:** defined conflict and reconciliation rules.
4. **Authentication and authorisation:** controlled access and permissions when the server is unavailable.
5. **Reporting:** a distinction between synchronised, pending, and conflicted data.
6. **User experience:** clear connectivity and synchronisation status without unnecessarily blocking sales.
7. **Testing:** slow connections, intermittent failures, duplicate requests, and corrupted queues.
8. **Support:** ways to diagnose synchronisation failures remotely.

What looked like "add an offline mode" was closer to rebuilding the transactional foundation. Ten interviews before development can be more valuable than months of implementation if they expose an assumption that would invalidate the product.

The market was not wrong

When users reject a product because it does not fit their environment, the easiest reaction is to blame the environment. The internet should be more reliable. Businesses should modernise their processes. Customers should adopt newer hardware.

Those statements do not rescue the product. Customers evaluate products based on the world they operate in today. If a store cannot depend on continuous connectivity, software that requires it transfers infrastructure risk to the customer.

From the customer's perspective, a manual workflow that continues during an outage can be safer than a modern system that becomes inaccessible. The resistance was not necessarily a failure of customer education. It was often the customer identifying a risk that

the product had underestimated.

The real competitor was not another POS

InclinePOS was not competing only with other software products. It was also competing with paper receipts, cash registers, spreadsheets, desktop applications, existing manual processes, and informal inventory practices.

These alternatives may be inefficient. They may lack centralised reporting and automation. But they often continue working under the customer's normal constraints. A notebook does not lose connectivity. A calculator does not need an API response. A desktop application with local storage may have weaker cloud capabilities, but it can continue processing sales when the internet disappears.

To replace an imperfect existing system, a startup must preserve the capabilities the customer considers non-negotiable. More features are not enough if the new product introduces a more dangerous point of failure.

What I should have done differently

The failure was preventable through better discovery and earlier testing of the riskiest assumption.

Observe the business during real operating conditions

Customer interviews are useful, but observation reveals what people forget to mention. I should have watched stores during busy sales periods, internet interruptions, shift changes, returns, inventory receiving, end-of-day reconciliation, and printer or device failures.

The goal would not have been to ask which features they wanted. It would have been to understand what could never be allowed to stop.

Identify operational invariants

An operational invariant is something the business must be able to do regardless of system conditions. For a POS, examples include recording a sale, calculating the total, accepting payment, producing a receipt, preserving the transaction, and reconciling the sale later.

These invariants should have shaped the architecture before dashboards, analytics, or secondary management features. Payment methods have their own constraints: a locally recorded sale does not mean an offline card payment has been authorised. The product needs a supported payment-provider flow and a clear fallback for each method.

Test the most dangerous assumption first

The biggest unknown was not whether I could build POS software. The bigger question was:

would target businesses accept a POS system that depended on internet availability?

A prototype, a workflow demonstration, and direct conversations with store owners could have exposed the issue early. A limited pilot under real network conditions would have made the risk harder to ignore.

Design for failure before designing for scale

Founders often think about scale in terms of users, requests, servers, and database load. InclinePOS also needed to work across unreliable environments, low-cost devices, different employee skill levels, and inconsistent operating practices. It needed a support plan when synchronisation failed. Resilience mattered before cloud scale.

Treat infrastructure constraints as product requirements

Connectivity, devices, printers, power, and local networks are part of the product if the product depends on them. A better requirements document would have included conditions such as:

- A cashier must be able to record and complete a supported sale without internet access.
- Transactions must remain durable after an application restart.
- Every pending transaction must have visible synchronisation status.
- Retrying a synchronisation request must not duplicate a sale.
- Conflicts must be recorded rather than silently overwritten.
- The system must provide a recovery process after device failure.

These are testable requirements. "Support offline mode" is not precise enough.

How I would architect it today

If I were designing InclinePOS again, I would begin with an offline-first transaction model rather than adding offline behaviour later. This is a proposed design, not a description of a rebuilt or tested InclinePOS system. The exact technology would depend on whether the application was web-based, desktop-based, or mobile-based.

Local-first transaction creation

Every sale would be created locally with a globally unique transaction ID, device ID, cashier ID, local timestamp, product and price snapshots, tax and discount calculations, payment details, and a synchronisation state. The sale would not depend on an immediate server response to be considered locally recorded.

A local record and a completed payment are separate states. Avoid storing sensitive card data in the POS unless the system is designed and approved for it; use the payment provider's supported flow and represent pending or failed payment honestly.

Durable local storage

For a web application, IndexedDB can store structured local data. A desktop application could use SQLite; a mobile application could use an embedded database appropriate to its platform. [1] [3] The important requirement is durability, not a particular technology.

Browser storage is best-effort by default and can be evicted or cleared; a request for persistent storage may be denied. [2] Test the chosen devices and browser settings, make storage failures visible, and plan backups or local replication for unsynchronised data. No local database can recover the only copy after a device is lost.

An append-only outbox

Unsynchronised operations would be placed in a local outbox. Each operation would move through explicit states: pending, synchronising, confirmed, failed, and conflicted. The user should be able to see whether data is safely synchronised.

Write the sale and its outbox entry atomically in local storage, so a crash cannot record one without the other. Authentication and cached permissions also need explicit expiry, revocation and recovery rules; a disconnected device cannot learn immediately that a user's access has been revoked.

Idempotent synchronisation

Every request sent to the server would include an idempotency key. If the application retried a transaction because the acknowledgement was lost, the server would recognise the operation rather than create a duplicate sale. Provider-supported idempotency has its own scope and retention rules, so check the specific payment service rather than assuming a key guarantees indefinite deduplication. [4]

Explicit conflict policies

Different data needs different conflict rules:

- Completed sales should be immutable; corrections should be recorded separately.
- Product details could use versioning.
- Inventory discrepancies should create reconciliation records.
- Administrative changes might require the latest server version.
- Price changes should not retroactively alter completed transactions.

"Last write wins" would be inappropriate for many financial operations because it can silently destroy information.

Auditability

Every important event should be traceable: who created the transaction, on which device, when it was created locally, when it reached the server, whether it was retried, whether it produced a conflict, and who resolved that conflict. This is useful for debugging and for trust with the business owner. Protect those records and avoid retaining unnecessary payment or personal data.

Failure-oriented testing

The test suite should simulate the connection disappearing during checkout, a lost acknowledgement after server processing, an application closing before synchronisation, multiple devices editing related records, repeated retries, incorrect device clocks, long offline periods, and a partially corrupted synchronisation queue.

A system is not offline-capable merely because it works after toggling Wi-Fi off for thirty seconds. It must be tested as a distributed system with delayed, duplicated, reordered, and conflicting operations.

The broader founder lesson

InclinePOS changed how I validate product ideas. I no longer begin only with whether the software can be built, which technology to use, or how quickly an MVP can launch.

I begin with more fundamental questions:

- What must never stop working?
- What assumptions does the product make about the customer's environment?
- Which assumption could invalidate the entire solution?
- What does the customer do when the system fails?
- Is the existing alternative less capable but more dependable?
- Which requirements would force an architectural rewrite if discovered later?
- Can the riskiest assumption be tested before building the full product?

These questions matter for products inside critical workflows, including payments, healthcare, logistics, education, manufacturing, field operations, retail, and financial systems. Reliability is part of the customer's willingness to trust the product.

Build for the customer's world

Founders build mental models of how a market works. Customers operate inside the actual market. The distance between those two realities is where products can fail.

InclinePOS taught me that elegant systems can still be based on incorrect premises. A

sophisticated solution can lose to paper, spreadsheets, or older desktop software when those alternatives better survive the customer's environment.

Before optimising the architecture, understand that environment. Before adding features, identify what cannot be allowed to fail. Before assuming users are resistant to change, consider whether they have identified a risk the product ignored.

*[Figure: image]***Code is not a business: what 20 paying customers taught me**: I built a POS SaaS, sold it to about 20 shops, then saw it solved the wrong problem. How I validate a SaaS idea now, and the checklist I use before coding. *[Figure: image]***Simple wins: designing for the least technical user in the room**: People pick software they enjoy over the longest feature list. A practical guide to designing SaaS for users who would rather not use software at all.

> **Key takeaways**
>
> - Test the assumption most likely to invalidate the product before building around it.
> - Identify which customer workflows must continue during an outage.
> - Treat offline transactions, synchronisation and recovery as architectural requirements.
> - Separate a locally recorded sale from an authorised payment.
> - Compare the product with the dependable manual workflow it must replace.

Questions readers ask

Why did InclinePOS fail?

InclinePOS depended on stable connectivity, an assumption that did not fit the operating conditions of the businesses it was intended to serve. Offline operation was an architectural requirement rather than a small feature to add later.

What does an offline-first POS need?

It needs durable local transaction storage, visible synchronisation status, idempotent retries and explicit conflict policies. Payment, security, device loss and recovery also need separate plans.

How can founders identify offline requirements earlier?

Observe the critical workflow under real operating conditions, identify what must continue during an outage and test the connectivity assumption before building the full product.

References

1. MDN Web Docs. IndexedDB API.

2. MDN Web Docs. Storage quotas and eviction criteria.
3. SQLite. Appropriate Uses For SQLite.
4. Stripe. Idempotent requests.

CITE AS

Gulzaib, M. (2026, August 18). The InclinePOS Failure: What Ignoring Offline Requirements Taught Me. Gul's Notebook. https://gulzaib.vercel.app/blog/the-inclinepos-failure-what-ignoring-offline-requirements-taught-me

© 2026 Muhammad Gulzaib. Quote short passages with a link and credit; please ask before republishing the full text.