Banking App Refactoring & Performance Optimization: Central African Consortium

An unstable banking app with 200,000+ users, stabilized and rebuilt for scale, funded entirely by the bug-fix budget it eliminated.

Book a Consultation

SERVICES

IT audit, Code refactoring, App performance optimization, Architecture Modernization, CI/CD Setup

INDUSTRY

Banking & Finance (Fintech)

CASE STUDY CATEGORIES

Finance, Fintech

THROUGHPUT AND RESPONSE TIME

10× / 5×

CRASH FREE IN PRODUCTION:

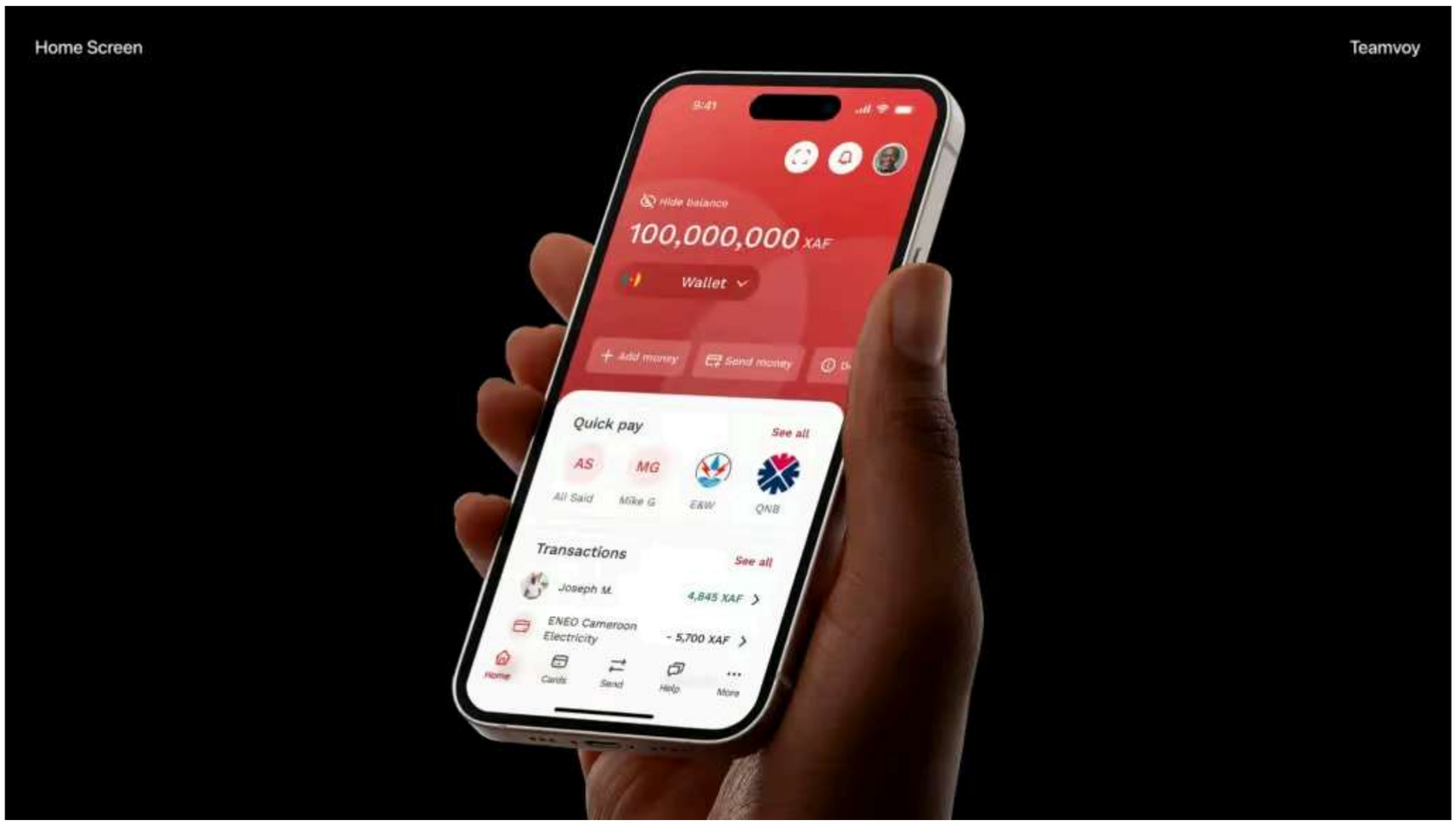
98%

Executive Summary

How Did A Central African Banking Consortium Turn An Unstable App Into A Platform Ready To Scale?

A leading financial services consortium in Central Africa — seven banks operating under one group — had launched a white-label banking app that other member banks could customize and adopt. The product worked: more than 200,000 users signed up. But under the surface, performance issues were capping transaction throughput, the app was crashing in production, and there was no monitoring, no logs, and no system-health tracking to explain why. Before the consortium could grow the user base or push transaction volumes higher, the foundation had to be rebuilt.

This case study walks through how Teamvoy stabilized the app, refactored the codebase, modernized the architecture, and put load-tested infrastructure under it — all in four months, with two backend developers and a DevOps expert. The result: 10× more requests per second, 5× faster response times, crashes eliminated, 70%+ test coverage on new code, and a refactoring program funded entirely by the 30% of the monthly development budget that used to be spent on bug fixes.



01. Our client

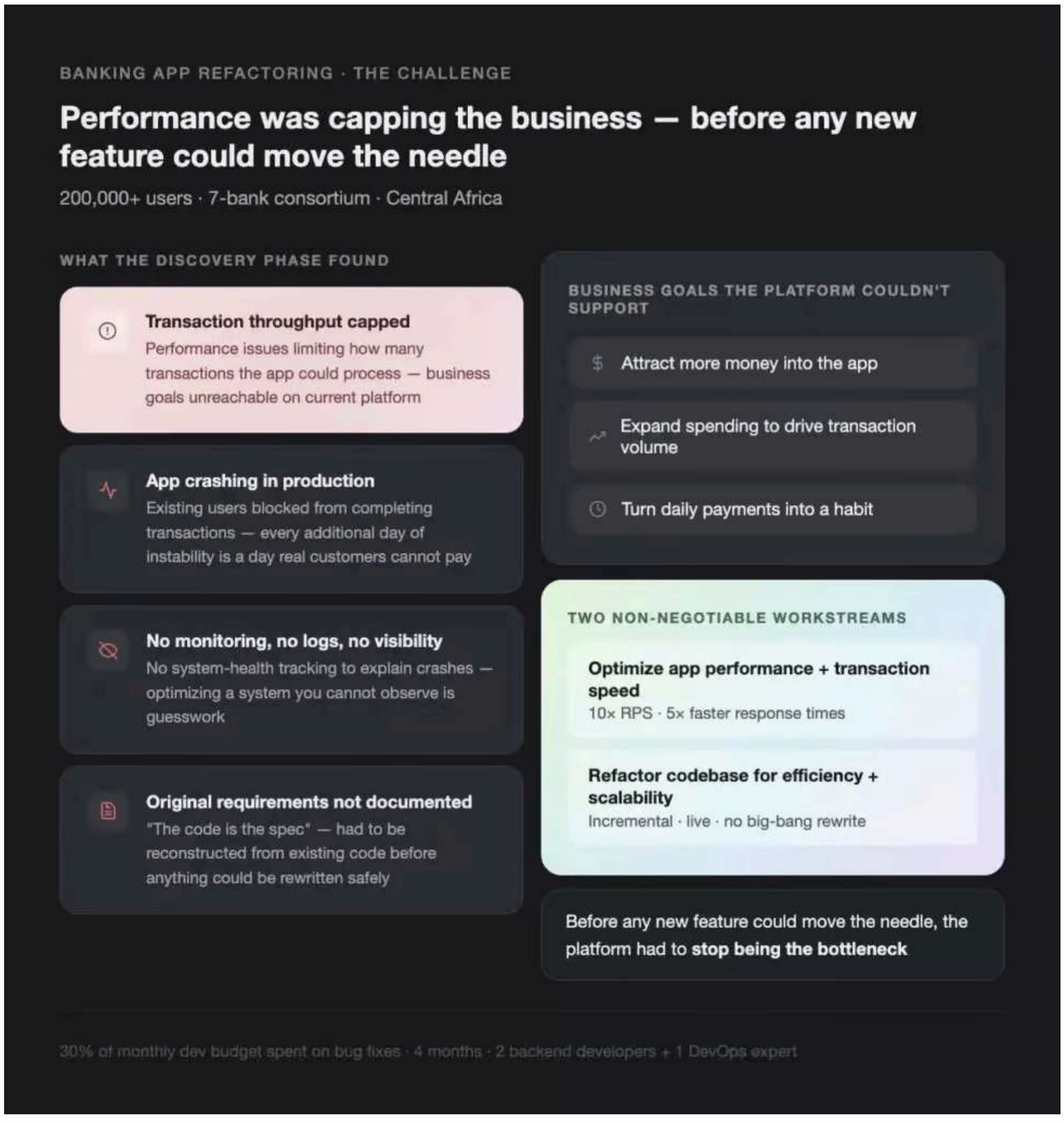
Who Is The Client And Why Did The Banking App Need To Scale?

The client is part of a leading financial services consortium in Central Africa made up of seven banks. To deliver a next-level online banking experience to both B2B and B2C customers, the consortium built a white-label mobile banking app that other banks in the group could brand and adopt. The product hit the market well: more than 200,000 users came on in a relatively short window.

That early traction reset the consortium's ambitions. The next phase was bigger — attract more money into the app, expand spending opportunities to drive transaction volume, and turn daily payments via the app into a habit instead of an occasional behavior. The product team had a clear roadmap for how to get there, but the engineering underneath the app could not support it. The client turned to Teamvoy on the strength of an existing engagement on another part of the solution, and on a reputation for handling exactly this kind of remediation work.

02. The Challenge

What Problem Does Banking App Refactoring Solve When Performance Is Capping The Business?



The client's product goals — more inflows, more spending, more daily transactions — were straightforward on paper. The discovery phase made it clear they were not achievable on the current platform. Performance issues were limiting how many transactions the app could process. The infrastructure was outdated. There was no monitoring, no logs, no system-health tracking. The app was unstable enough that even existing users could not rely on it as intended.

In other words: before any new feature could move the needle, the platform had to stop being the bottleneck. Two technical workstreams were non-negotiable — optimize app performance and transaction speed, and refactor the codebase for efficiency and scalability — and they had to happen against a complication that is common in older fintech codebases but still costly: the original system requirements were not documented. They had to be reconstructed from the existing code before anything could be rewritten safely.

For a banking product, this is the worst class of problem to inherit. Every change risks regressions, and "the code is the spec" is not a position any compliance officer wants to be in.

03. Approach

What Is Incremental Refactoring, And Why Did It Fit A Live Banking App?

Banking app refactoring is the disciplined rewrite of an existing financial product's codebase and architecture — preserving regulatory behavior and user-facing functionality while replacing what cannot scale underneath. The category ranges from cosmetic cleanups to full backend rebuilds with new infrastructure, monitoring, and CI/CD pipelines.

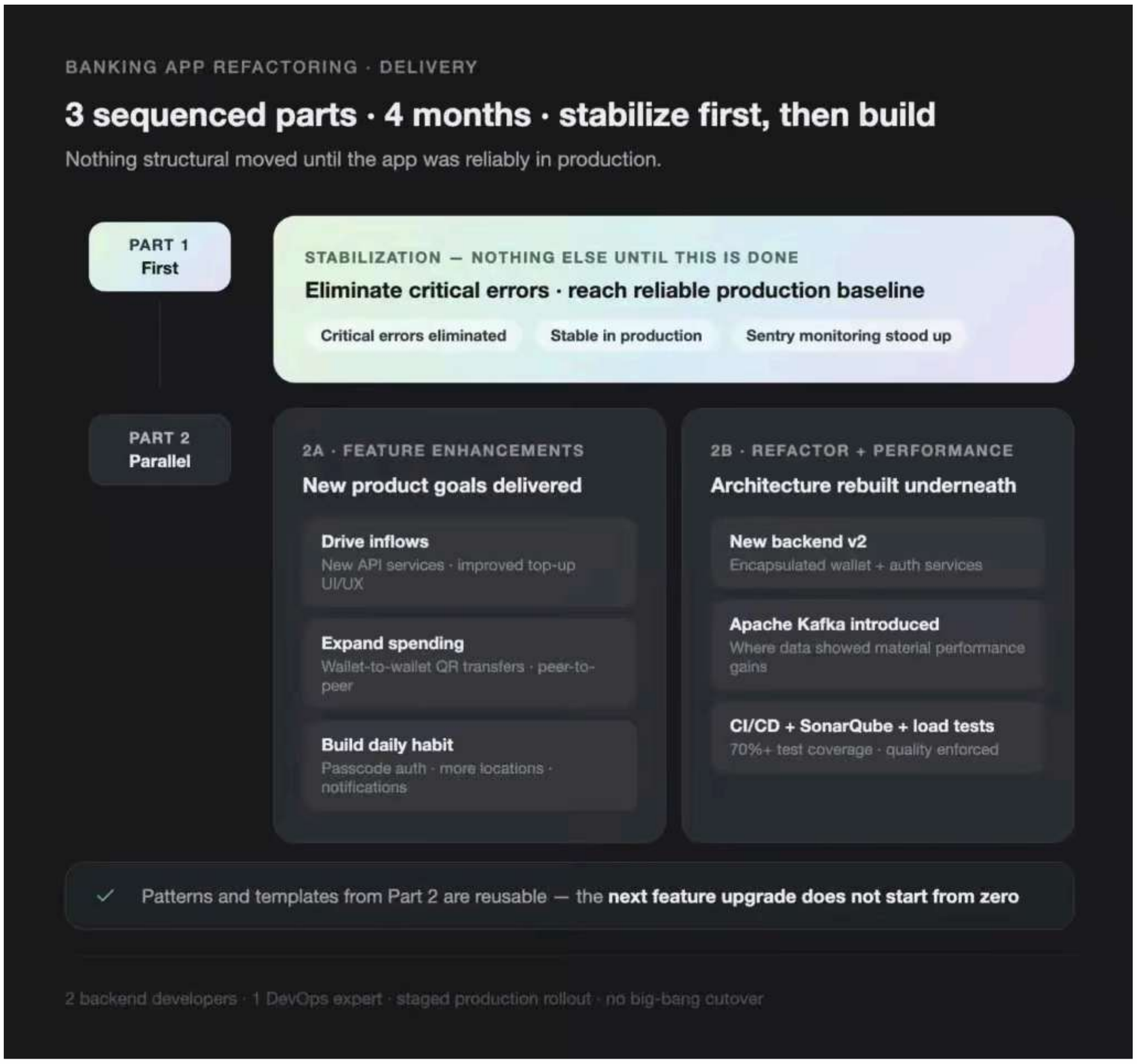
For this consortium, the requirement was specifically a refactor that could happen on a live app serving 200,000+ users without an extended stabilization period in production. A single big-bang rewrite was off the table — every additional day the app was unstable was a day real customers could not pay. The fit was an incremental approach: break the work into smaller parts, ship them in sequence, and let each stage stabilize before the next one begins.

The deeper reason this worked is that it draws a clean line between stabilization and modernization. The team did not try to refactor a broken system. It made the system reliable first, then refactored against a foundation that could be trusted. For a banking product, that ordering is the difference between a controlled program and a series of production incidents.

What We Did

How Did Teamvoy Stabilize, Refactor, And Modernize The Banking App?

The engagement ran in three sequenced parts — stabilization first, then feature enhancements and architectural refactoring in parallel — over four months.



Part 1 — Stabilization. Before any major upgrade, the app had to reach a reliable baseline. The team eliminated critical errors, brought the app to stable performance in production, and only then moved forward. Nothing else shipped until that foundation was in place.

Part 2(a) — Feature enhancements. With the app stabilized, the team turned to the product goals. Planning covered architecture and database structure, requirements reconstructed from the existing code, backend tasks for the new architecture, a DevOps workstream for new infrastructure, and database migration scripts. Development then delivered against the three business goals directly: a new version of API services and a clearer top-up UI/UX to drive inflows; wallet-to-wallet QR transfers to expand spending and enable peer-to-peer transactions; passcode authentication, broader top-up and spending locations, and new notifications to make daily app use habitual.

Part 2(b) — Refactoring and performance optimization. Running in parallel with the feature work, the team rebuilt what was holding the platform back: a new backend (v2) for the upgraded functionality, encapsulated wallet and authentication services for flexibility and scale, a new architecture sized for higher load, old code rewritten and folded into the new architecture, Sentry stood up as a fast monitoring fix ahead of more sophisticated tooling, and Apache Kafka introduced where it materially improved performance. Functional and load tests ran throughout, with SonarQube integrated to track test coverage and code quality.

Once the upgraded functionality was production-ready, it rolled out in stages. The patterns and templates created during this phase are reusable — the next feature upgrade does not start from zero.

Tech Stack:

Which Technologies Power The Refactored Banking App?

- Backend v2 — a new backend rebuilt to support the upgraded functionality and a scalable architecture.
- Apache Kafka — introduced where it materially improved throughput and decoupled high-load paths.
- Sentry — code monitoring stood up as a fast first step, ahead of more sophisticated observability tooling.
- SonarQube — test coverage and code quality assessment integrated into the development loop.
- CI/CD pipeline — newly implemented to make every deploy repeatable and every regression catchable.
- Load testing framework — used to validate throughput and response-time targets before production rollout.

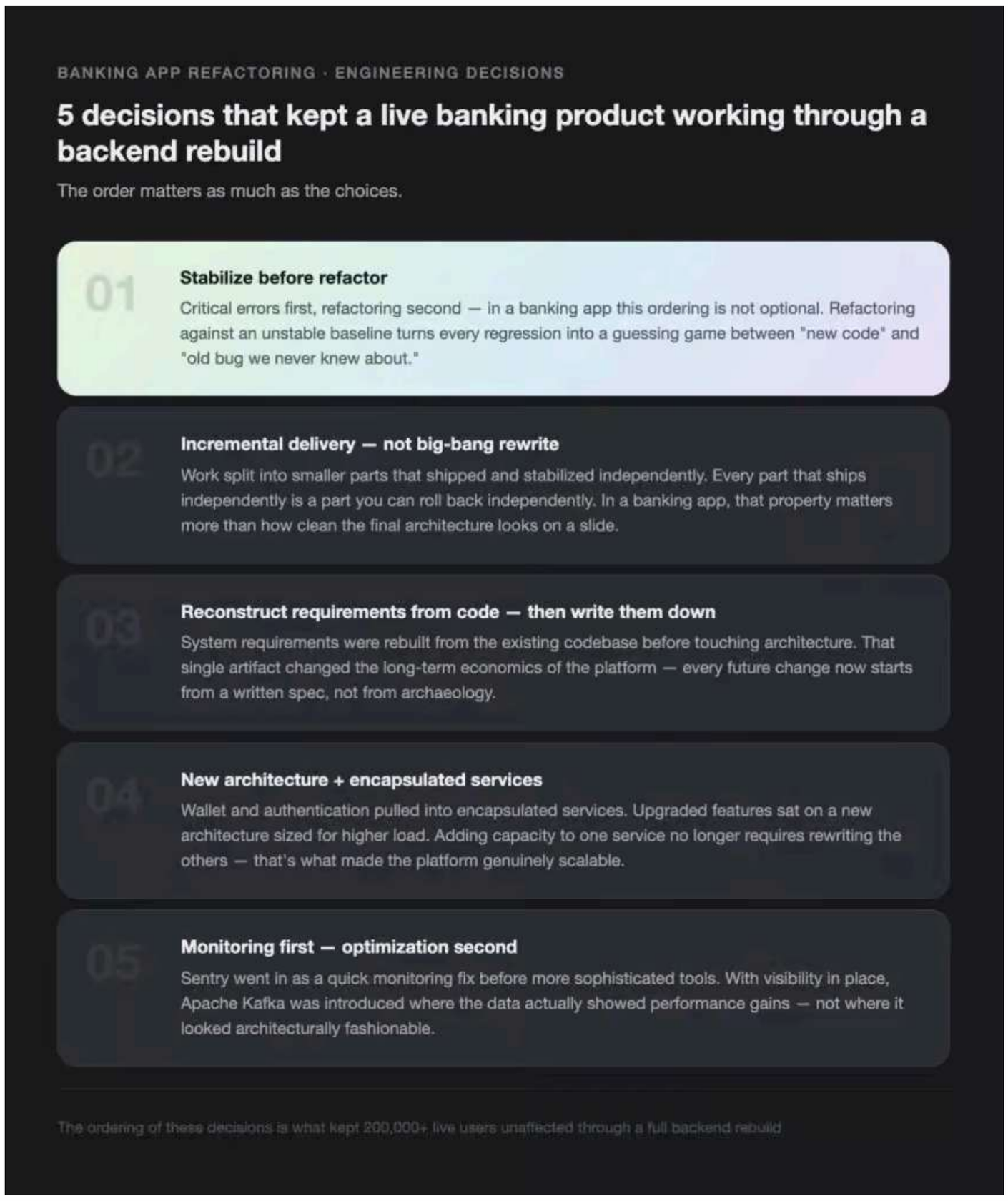
Key Features:

Which Features Define A Production-Ready, Refactored Banking App?

- Wallet-to-wallet QR transfers — enter an amount, generate a QR code, have another user scan it, and the funds move instantly, peer-to-peer.
- New API services and an improved top-up UI/UX, making wallet funding more visible and intuitive to drive inflows.
- Passcode authentication — faster login that lowers friction on daily payments.
- Expanded top-up and spending locations, plus targeted notifications that nudge customers toward frequent app usage.
- Encapsulated wallet and authentication services, designed to be flexible and scalable as the consortium adds member banks.
- CI/CD-backed release flow with load-tested infrastructure, code monitoring, and 70%+ test coverage on new code.

Which Engineering Decisions Made The Refactor Reliable Under A Live User Base?

Five decisions shaped the program's reliability — and they are the same ones that kept a live banking product working through a backend rebuild.



Stabilize before refactor. Nothing structural moved until the app was reliably in production. Critical errors came first, refactoring came second. In a banking app, this ordering is not optional — refactoring against an unstable baseline turns every regression into a guessing game between "new code" and "old bug we never knew about."

Incremental delivery instead of a big-bang rewrite. The work was split into smaller parts that could ship and stabilize independently. That shortened the in-production stabilization window after each release and let specific improvements land sooner, instead of holding everything for one large cutover that no compliance team would have signed off on anyway.

Reconstruct requirements from the code, then write them down. System requirements were not documented at the start of the project. The team rebuilt them from the existing codebase before touching architecture, and produced well-documented product requirements as a deliverable. That single artifact changed the long-term economics of the platform — every future change now starts from a written spec, not from archaeology.

New architecture for upgraded features, encapsulated services underneath. Wallet and authentication were pulled into encapsulated services, and the upgraded features sat on a new architecture sized for higher load. That separation is what made the platform genuinely scalable — adding capacity to one service no longer requires rewriting the others.

Monitoring first, optimization second. Sentry went in as a quick code-monitoring fix before more sophisticated tools, because optimizing a system you cannot observe is guesswork. With visibility in place, Apache Kafka was introduced where the data actually showed performance gains — not where it looked architecturally fashionable.

05. Impact

What Impact Did Refactoring Have On The Banking App And The Business?

The project's measured impact ran well past the original feature scope. Load testing came back with 10× more requests per second and 5× faster response times. Crashes and failures that had blocked existing users were eliminated. A newly implemented CI/CD flow turned deploys from events into routine. New code shipped with 70%+ test coverage, with SonarQube enforcing quality on everything that followed.

On the business side, the load-testing results gave the consortium a direct lever on cost: lower server and maintenance bills, and headroom to scale into the higher transaction volumes the product roadmap depended on. Documented requirements and updated test cases also moved the asset from "a product that runs" to "a product that can be handed to another team without losing institutional memory."

Qualitative Results At A Glance

- 10× more requests per second and 5× faster response times after load testing — measured against the pre-refactor baseline.
- App crashes and failures that had blocked existing users were eliminated; the app is stable in production.
- New CI/CD flow turned releases from a risk event into a routine operation.
- 70%+ test coverage on new code, with SonarQube enforcing code quality going forward.
- Product requirements were reconstructed from the existing code and properly documented — the platform now has a written spec.
- Load-testing headroom let the consortium reduce server and maintenance costs while opening the path to higher transaction volumes.

The economics of how the refactor was funded are worth a paragraph on their own. Before the engagement, 30% of the client's monthly development resources went to bug fixing. Once the app was stabilized, that work disappeared. The entire refactoring program was funded out of that same 30% — no incremental budget, no business case to defend, just a reallocation of money that was already being spent on symptoms. The broader payoff was operational: the platform stopped being the constraint on the product roadmap, and the next phase — a hybrid cloud-based architecture, also handled by Teamvoy — became possible.

What Should Banking And Fintech Teams Know Before Refactoring A Live App?

A few takeaways generalize beyond this engagement and apply to anyone weighing a refactor on a live financial product with real users behind it.

Stabilize first, refactor second. It is tempting to do both in one motion. Don't. A refactor on top of an unstable system makes every regression unattributable, and in banking software that is the worst class of debt to take on. Get the app reliable, then change what it is made of.

Reconstruct requirements before touching architecture. If the spec lives only in the existing code, you do not have a spec — you have a guess. Write it down before you change it. Future audits, future vendors, and future engineers will all be downstream of that one document.

Make the refactor pay for itself. Most live financial products carry a recurring bug-fix tax that is large enough to fund the rewrite that would eliminate it. Audit that line item before asking for a new budget. The case for refactoring is almost always already inside the existing development spend.

Incremental beats big-bang on regulated products. Every part of the rewrite that ships independently is a part you can roll back independently. In a banking app, that property matters more than how clean the final architecture looks on a slide.

Conclusion

Where Should Banking And Fintech Teams Start With App Refactoring?

For this Central African consortium, refactoring the banking app was less about chasing a cleaner codebase and more about removing the constraint that was capping the business. The stabilization + incremental refactor + new architecture sequence turned an unstable product serving 200,000 users into a platform with 10× the throughput, 5× faster response times, and the headroom to actually pursue the consortium's growth plan.

If you are evaluating a banking app refactor, the most important question is not "which framework or queue should we adopt?" — it is "what is the bug-fix tax we are already paying, and what would the platform look like if that money funded a rebuild instead?" The answer is usually the case for refactoring, written in numbers the finance team already trusts.