ChatGPT

Zanari Technical Feasibility and Architecture

Executive Summary: Zanari's design leverages existing Kenyan mobile-money and card APIs to minimize licensing delays. Safaricom's M-PESA **Daraja** API (REST/OAuth2) provides endpoints for user payments (Lipa na M-PESA / STK Push), merchant pull (C2B), disbursements (B2C), and transfers (B2B) [1] [2]. Likewise, Airtel Money offers OAuth2-based REST APIs via its developer portal [3] [4]. Telkom's T-Kash lacks public APIs, so integration would use a payment gateway or aggregator (e.g. Interswitch's PayWay supports TKash alongside M-PESA/Airtel [5]). For debit cards, partnering with a card-issuing service (e.g. Miden, Sudo) is optimal, since these handle PCI-DSS and settlement [6] [7]. The recommended architecture is cloud-native microservices (e.g. Node.js/Go backends, PostgreSQL, Kubernetes) with React Native on mobile [8] [9]. Strict security is essential: encrypt data in transit and at rest, use OAuth2/JWT for auth, and comply with Kenya's Data Protection Act and CBK guidelines [10] [11]. Automated "save-as-you-spend" features can be built with event-driven triggers on each payment (e.g. "round-up" transfers) [12]. We propose an MVP in 0–6 months (basic wallet, card funding, one mobile-money integration), then roll out group savings and analytics (Phase 2), and finally scale-out and add innovations (Phase 3). Risk mitigation includes relying on established platforms (cloud services, card issuers) and designing for compliance from day one.

1. Integration Requirements Analysis

M-PESA (Safaricom Daraja)

Safaricom's **Daraja API** is the standard way to integrate with M-PESA. Daraja is a RESTful API (replacing the legacy SOAP G2 API) that is publicly accessible on the internet [2]. To use it, Zanari must register on the Safaricom developer portal and create an application, obtaining a **ShortCode** (till number or paybill) and OAuth credentials (Consumer Key/Secret). The app then calls the OAuth endpoint (e.g. `https://sandbox.safaricom.co.ke/oauth/v1/generate?grant_type=client_credentials`) to receive a time-limited access token [1]. With this token, the app can call Daraja's endpoints:

- **C2B (Customer-to-Business)** – For user payments into Zanari's merchant account (e.g. User pays Zanari's PayBill/Till). You must register confirmation/validation URLs to receive asynchronous callbacks.
- **Lipa na M-PESA (STK Push)** – Zanari initiates an STK Push to the user's phone for checkout. The user enters their PIN, and upon success Zanari's server gets a payment confirmation webhook [2].
- **B2C (Business-to-Customer)** – Disbursement API for pushing funds to users' M-PESA wallets (e.g. sending saved funds out to a user's phone).
- **B2B (Business-to-Business)** – For transfers between Zanari's M-PESA accounts or between Zanari and partner businesses.
- **Balance/Status Queries** – APIs to check account balances or transaction status.

Safaricom provides a sandbox environment for testing. Production use requires a live short code and may involve commercial contracts. Transaction fees are charged per the M-PESA tariff schedule (typically a small % of the amount, capped by law). As one developer writes, "the Lipa na M-PESA online service…triggers an STK push to the user's phone…When the user enters the PIN…the user's M-PESA account is debited, and you

receive a webhook with details of the transaction" [2]. In summary, **M-PESA integration is robust and well-documented**: use OAuth2 for authentication, and Daraja's REST endpoints for all payment types.

Airtel Money Integration

Airtel Money (Airtel Kenya) provides a developer API via the *Airtel Africa* platform. The flow is similar to M-PESA: developers create an account on developers.airtel.africa and register an application, enabling "Collection" and "Remittance" APIs. Airtel's API uses OAuth 2.0 client credentials. After obtaining a `client_id` and `client_secret`, the app POSTs to the Airtel OAuth token endpoint (e.g. `https://openapi.airtel.africa/auth/oauth2/token`) to get a Bearer access token [3]. With the token, the app can call endpoints like `POST /merchant/v1/payments/` to initiate a payment. For example, a USSD push ("collect money") request is submitted with fields for amount, subscriber phone, and a unique reference. Upon calling this API, "the consumer will be asked to authorize the payment by entering their Airtel Money PIN... After authorization, the transaction will be executed" [4]. In practice, this works like an STK push: the user confirms on their phone and Airtel calls back to Zanari's specified URL with the result. The API covers merchant pull (charging user), and there are also remittance endpoints for paying out (similar to M-PESA B2C). The sandbox uses `openapiuat.airtel.africa` and production uses `openapi.airtel.africa` [3]. Overall, Airtel Money integration is fully supported via REST/OAuth2, with developer docs and sandbox environments in place.

Telkom T-Kash Integration

Telkom Kenya's **T-Kash** mobile money platform is newer and does not have an open public API like M-PESA or Airtel. Telkom's internal solution (built on WSO2) suggests they use an API gateway, but detailed developer docs are not publicly available. In practice, fintechs integrate T-Kash by partnering or using a payment aggregator. For example, **Interswitch's PayWay** (a Nigerian-origin payment gateway operating in Kenya) supports MPESA, Airtel Money, and T-Kash [5]. This means if Zanari integrates with Interswitch's API, it can automatically handle all three networks. Alternatively, local fintech aggregators (such as MFS Africa or others) may provide a unified API. If direct integration with Telkom is required, Zanari would likely need a formal partnership and sandbox credentials (as hinted by unofficial guides). In summary, T-Kash integration is *possible* but less straightforward; using a multi-operator payment gateway is the most pragmatic approach [5].

Debit Card Implementation Options

To offer debit card payments, Zanari needs to either partner with a bank/card issuer or use a card-issuing fintech platform. In Kenya, options include:

- **Licensed Banks or Card Programs:** The traditional route is to collaborate with a bank that can issue Visa/Mastercard debit cards linked to Zanari's wallet (or a co-branded card). For example, KCB M-PESA cards exist, but a fintech program would require negotiating a BIN sponsorship and compliance with the bank's issuing platform.
- **Fintech Card Issuers:** Several fintechs provide API-driven card issuance. For instance, *Miden* offers a card API enabling businesses to issue KES or USD debit cards to customers [6]. Miden handles local settlement and compliance (PCI-DSS, CBK regs) so Zanari can control cards via API calls. Similarly, *Sudo* (open platform) advertises the ability to "programmatically create, manage, and distribute both

physical and virtual cards" with advanced controls [7]. These platforms often offer features like instant virtual card creation, spending limits, and card freezing via API.

- **Global Fintech Platforms:** Companies like Rapyd or Stripe Issuing (if available in Kenya) offer global card APIs, but may not support KES or require KYC alignment.
- **Cryptocurrency-Linked Cards:** Some crypto wallets (e.g. Bitnob) offer USD virtual cards funded by crypto or mobile money [13], but this adds crypto dependency and FX risk.

Given Zanari's desire for immediate launch *without obtaining licenses*, using a card-issuing partner is best. These partners bear the regulatory burden (e.g. PCI-DSS, data protection). Technically, Zanari would use the partner's API to create cardholders and cards (getting card PANs or tokens). Users could then fund their wallet by funding the card or use the virtual card directly at merchants online. One can issue virtual card numbers via API instantly [6], and even produce physical cards on demand. In short: **debit card functionality is viable by piggybacking on licensed issuers**. For example, a future version of Zanari could instantly create a virtual Visa/Mastercard (in KES/USD) via Miden's API [6] or Sudo's API [7], enabling user spending anywhere cards are accepted.

2. Technology Stack Recommendations

Backend Architecture

A scalable, **microservices** architecture is recommended. Modern fintech stacks favor cloud-native, containerized services. According to industry analysis, "Node.js, Go, [and] Python…are still top picks for performance and scalability" [8]. Using Node.js (with Express/Koa) or Go can provide fast I/O for payment processing. For example, one could implement payment integrations and business logic as separate services (e.g. an *MPESA Service*, an *Airtel Service*, a *Card Service*, etc.), each with its own REST API. These services can be containerized (Docker) and orchestrated via Kubernetes on a cloud platform (AWS EKS, Azure AKS, or similar). This enables horizontal scaling: if traffic spikes (e.g. end-of-month saving periods), we can spin up more pods. Key backend components: a web API gateway (securely proxying to services), an authentication service (OAuth2/JWT issuance), and microservices for wallets, transactions, users, savings rules, and analytics.

Data storage should use a robust DBMS: a relational database like PostgreSQL (for ACID safety on transactions, e.g. saving rules and group contributions) with read replicas, or a scalable NewSQL (e.g. CockroachDB) if necessary. A Redis cache can improve performance for frequent lookups (rate-limiting, session data, etc.). For third-party integrations (card issuance, SMS, notifications), backend services will communicate via REST/HTTPS.

Infrastructure-as-Code (Terraform/ARM templates) should manage all resources. Logging/monitoring (ELK stack or Prometheus/Grafana) is mandatory. In essence, **cloud-native microservices** in Node/Go/Python, with Kubernetes and managed DBs, will meet performance and resilience needs [14] [8].

Mobile and Web Frontends

Zanari will have both mobile apps (Android/iOS) and a web interface. The user has indicated React Native for mobile, which is sensible: it allows a single JavaScript codebase for both platforms while accessing native device features (camera, NFC, etc.) if needed. React Native performance is near-native and has strong community support. For the web app or admin portal, a React (or Next.js) frontend is appropriate, enabling

code reuse (React skills) and a PWA if desired. An alternative is Flutter (uses Dart), but given the team's React knowledge, React Native/React is best.

Using **React Native** means that Zanari can quickly iterate UI/UX and share some logic with React web components. If a fast MVP is needed, the technical literature suggests stacks like "Next.js (web) / React Native (mobile)" with a serverless or managed backend for rapid development [15]. For example, Phase 1 could even use Firebase/Supabase for backend/user auth to expedite launch [15] (serverless DB and functions), then migrate to custom services later. However, even with Firebase, the mobile code remains React Native.

Infrastructure & Cloud Services

A **cloud platform** with African region support is advised. Azure has a Kenya Central region (Nairobi), and AWS has South Africa (Cape Town) – either can satisfy data residency and low latency. Docker/Kubernetes should run in such a region. Use managed services where possible: e.g. AWS RDS (Postgres), AWS SQS/Kafka for queues, Amazon SNS for push notifications, etc. Kubernetes (EKS/AKS) allows rolling updates and high availability. Continuous Integration/Deployment (CI/CD) pipelines (e.g. GitHub Actions, ArgoCD) will automate builds and deployments.

For scalability, leverage auto-scaling groups and horizontal pod autoscaling based on metrics (CPU, queue backlog). Use a CDN for static assets. For session management or caching, Amazon ElastiCache (Redis) can be used. It's also prudent to include offline caching (on-device) and fallbacks (see below) for poor connectivity.

Financial cloud services: If available, use PCI-compliant services (e.g. AWS Lambda with secure VPC, Azure Confidential Compute). Because Kubernetes nodes must be PCI-compliant if card data touches them, it may be simpler to let the card API handle CVV/CVC and only store tokens on our side.

Data & Analytics Infrastructure

To power insights and personalization, set up a **data pipeline**. All transactions and user activities can be streamed via an event queue (e.g. Kafka or Kinesis). Downstream, load data into a data warehouse (e.g. AWS Redshift, Snowflake, or Azure Synapse) or a data lake (S3/Delta Lake) for analytics. Use ETL/ELT (e.g. Airflow jobs or AWS Glue) to transform and aggregate data (daily spend totals, goal progress, etc.). For real-time monitoring (fraud or anomalies), use stream processing (Kafka Streams or AWS Lambda).

Build reporting dashboards (e.g. Metabase, Tableau) for business insights. Machine learning models (TensorFlow/PyTorch) can be applied on this data to predict churn or recommend savings goals. The tech stack guide recommends planning for AI/ML with Go or Python and frameworks like TensorFlow [16]. Ensure all analytics respect data privacy (e.g. anonymize PII when needed).

3. Security Architecture Design

Encryption and Data Protection

All data in transit must use TLS (HTTPS) with strong ciphers. Sensitive data at rest (PII, account balances) should be encrypted using AES-256 or similar (e.g. AWS KMS). Passwords must be hashed (bcrypt/Argon2).

Card PANs should **never** be stored plaintext; use tokenization via the card provider or PCI-compliant vault. Regularly rotate encryption keys and tokens.

Kenya's **Data Protection Act (2019)** imposes strict safeguards. It requires explicit user consent for collecting personal data, and grants users rights (access, deletion, portability). The Act also empowers regulators to mandate **data localization** for strategic data [11]. In practice, Zanari should host user data on Kenyan (or equivalently-protected) servers and ensure compliance by design (e.g. privacy by default, data encryption, and breach notification procedures). Like GDPR, data processing must be lawful, and privacy notices must be clear. App features (profile data, KYC documents) should have opt-in and allow users to export or erase their data.

Authentication and Authorization

Implement a robust auth system. For users, use token-based auth (e.g. JWT or OAuth2 Resource Owner Password Flow). Include multi-factor authentication (SMS/OTP or authenticator app) for sensitive operations (transfers). Consider integrating with national ID verification or third-party KYC APIs for onboarding (e.g. verifying an ID number with government records).

For service-to-service and API access, use OAuth2 client credentials. All internal APIs should validate tokens and roles (RBAC). Use industry-standard libraries (Passport.js, Keycloak, etc.). Monitor for brute-force or anomalous login attempts.

Compliance and Best Practices

Financial apps are a high-security target. Follow OWASP Top 10 (prevent injection, XSS, CSRF, etc.). Conduct regular code audits and penetration testing. Use secure coding frameworks that mitigate common issues (e.g. parameterized queries).

PCI DSS is mandatory if Zanari handles card data. To simplify, offload as much as possible to the card issuer and **never store CVV or magnetic stripe data**. Use a PCI-compliant merchant processor for card funding/purchases. The industry guides recommend cloud services that offer PCI templates and compliance controls [17]. For example, AWS/Azure compliance centers can be leveraged.

Operational security: deploy firewalls and intrusion detection (e.g. AWS GuardDuty), and maintain an incident response plan. All transactions should be logged immutably; use a WORM (write-once, read-many) log or append-only ledger (e.g. blockchain-like audit log) for critical events (savings withdrawals, deposits) to meet audit requirements. The logs must be retained per CBK rules. Regularly patch dependencies and perform vulnerability scans on containers.

A risk-based approach is needed for AML/KYC: monitor transaction patterns for money laundering (sudden large sums, repeated small deposits). Compliance with the Proceeds of Crime & Anti-Money Laundering Act (POCAMLA) means verifying user identity (e.g. national ID, phone SIM registration) and keeping CDD records. Technical tools (like automated transaction monitoring) should flag anomalies.

Regulatory Considerations

Beyond data protection, CBK regulates payments via the National Payment Systems Act. While Zanari itself isn't a bank, providing wallet and money transmission services may classify it as a payment service provider. However, by integrating via Safaricom/Airtel (who hold PSP licenses), Zanari can operate under their umbrella (similar to how many cashback apps work). Zanari should still engage legal counsel to ensure no licensing breach.

In summary, **security and compliance** must be engineered in from the start: encryption, OAuth2, PCI-ready infrastructure, audit logging, and adherence to Kenyan law [10] [11]. As one analyst notes, Kenyan fintechs must align with CBK rules, PCI-DSS, and data protection requirements to avoid delays [10].

4. Automated Savings Engine

To implement "save-as-you-spend," we recommend an **event-driven rule engine**. The core idea is: every time the user spends money (via card or wallet), a predefined rule triggers a transfer of a small amount into their savings. For example, a common pattern is **"round-up" savings**: if a user spends KSh 93 on coffee, the transaction is rounded up to KSh 100 and the extra KSh 7 is automatically saved [12]. Another rule might be "save 10% of each purchase."

Technically, when a payment is confirmed (from the payments service), an event is emitted (message queue or webhook). A dedicated **Savings Service** subscribes to this event. The service retrieves the user's active rules from a database, calculates the savings amount, and then posts a fund-transfer transaction to the user's savings sub-wallet or separate account. This can be implemented as a microservice using a lightweight rules engine or even simple business logic in code. For real-time performance, a message broker (Kafka, RabbitMQ, or AWS SNS/SQS) decouples the payment and savings services.

For persistence, use a "savings ledger" in the database to record each triggered save (with references to the original spend). The Savings Service should handle edge cases (rounding errors, insufficient funds). Users can configure rules (percentage, round unit, daily/weekly cap) via the app settings. Over time, these micro-savings accumulate towards goals. The concept of "round-up savings" is well-known in fintech: it quietly builds balance without user effort [12]. We should also offer manual save options (like a "Rainy Day" boost) that directly call the same transfer function.

For analytics, track the effectiveness of rules (e.g. total saved per user) and use that for recommendations. If using an event-driven approach, ensure idempotency (duplicate payment events should not save twice) by storing processed event IDs.

In summary: an **event-triggered microservice** approach ensures that every qualifying spend automatically generates a corresponding save, implementing the "save effortlessly as you spend" promise.

5. Scalability & Performance

Zanari must support millions of users and high transaction volumes. Key considerations:

- **Microservices & Containers:** As noted, using Kubernetes (EKS/AKS) allows us to scale services individually. For instance, if the **payments service** experiences bursts, we scale only that pod. Stateless services (no local session state) facilitate horizontal scaling. Container images should be as slim as possible, and use auto-scaling groups based on load.

- **High-Throughput Databases:** Transaction records may grow very fast. Partition (shard) the database if needed (by user region or account) and use replication. Use read-replicas for heavy read loads (e.g. showing transaction history). Consider separating OLTP (transaction DB) from OLAP (analytics DB) to optimize performance.

- **Caching & Queues:** Deploy Redis or Memcached for caching frequently accessed data (user profiles, exchange rates, nonces). Use a durable message queue (Kafka/RabbitMQ/SQS) for asynchronous tasks (like sending SMS receipts, running savings jobs).

- **Cloud-Native Infrastructure:** The chosen cloud (AWS/GCP/Azure) should be one with African zones. Anecdotal reports say >90% of fintechs go cloud-native for agility [14]. We will use services in a region close to Kenya. Content can be distributed via a CDN for performance.

- **Resilience:** Use multiple availability zones. Design for failure: if one server goes down, load balancer should route around it. Kubernetes will auto-restart failing pods. We should also consider multi-region read replicas for disaster recovery if data law allows.

- **Technology Choices:** Based on tech reviews, a "modern cloud-native stack" might be React/TypeScript front-end, Node.js/Go backend, PostgreSQL + Redis, on AWS with Kubernetes [18]. These components are battle-tested. Node.js or Go will handle concurrent request volumes well, especially if we deploy it in a cluster. Python (e.g. Django/FastAPI) could also be used, but Node.js aligns with React and has a vast ecosystem.

- **Monitoring & Logging:** Implement distributed tracing (OpenTelemetry) so any performance bottleneck is visible. Use Prometheus/Grafana to monitor latency, error rates, and DB performance. Alert on thresholds (e.g. queue length, CPU usage).

In short, the stack should be built for **elasticity**: auto-scale on demand, employ caching, and use cloud provider tools for load balancing and CDN. Fintech tech surveys emphasize using Kubernetes/Terraform and modern languages to meet performance targets [14] [8].

6. Regulatory & Compliance Implications

Kenyan regulations will strongly influence architecture:

- **Financial Licenses:** Central Bank of Kenya (CBK) currently requires payment service providers to be licensed. Zanari's model (an app letting users hold wallet balances and spend via card/M-PESA) may

be considered a payment institution or "electronic money issuer." However, by tying directly into Safaricom/Airtel (their e-money) and using partners' card networks, Zanari might operate under their umbrella. Legally, Zanari must consult regulators to determine if any license is needed. Architecturally, we should prepare to segregate user float (perhaps keep only user fund balances as "custodial" in trust accounts via a bank, rather than Zanari's balance).

- **Data Localization:** The Data Protection Act 2019 empowers authorities to mandate certain data be stored in Kenya [11]. To comply, we should deploy core databases (especially PII) within Kenya's data centers/regions (e.g. Azure Kenya or AWS Africa Cape Town with encryption). Cross-border transfer of personal data would require safeguards (like SCCs) if any overseas processing is needed.

- **Data Protection & Privacy:** We must implement all Data Protection Act requirements: user consent flows, privacy notices in-app, secure data storage, and a Data Protection Officer. Features like letting users delete their accounts must truly erase personal data (or anonymize it). Personal data of children or sensitive categories would need special handling.

- **PCI-DSS & Card Data:** Since virtual card generation is involved, PCI-DSS compliance is implied. If Zanari does not touch raw PAN/CVV, the risk is lower, but environment (e.g. servers, networks) must be hardened. Regular PCI audits may be necessary. Using a PCI-certified partner for card issuance (as recommended) pushes most burden to them [10].

- **KYC/AML (POCAMLA):** Payment laws require KYC for wallet users. Expect to collect national ID, proof of address, and do screening for money laundering (e.g. PEP list, sanctions). Technically, integrate with KYC/KYB services and record verifications. Transaction limits and monitoring flows must be built in.

In sum, the architecture must include safeguards for **CBK's financial regulations and Kenya's Data Protection**. This affects choices like deployment region, encryption, logging, and user workflows (for consent and data access). We will maintain audit trails to demonstrate compliance.

7. Automated Savings Engine (Detailed)

The automated savings engine ("money jar") will be implemented as follows:

- **Rule Configuration:** The user can set rules in the app (e.g. "round up each spend to the nearest 50 KSh", or "save 10% of each purchase", or "save KSh 1000 every Friday"). These rules are stored in the backend database, linked to the user's account and wallet.

- **Event Trigger:** Whenever the user spends money (via card or app payment), the backend payments service completes the transaction and emits an event or calls the Savings Service API. For example, after a successful transaction, a message is placed on a queue (e.g. Kafka topic "tx_completed").

- **Savings Microservice:** A dedicated **Savings microservice** listens to the queue. Upon receiving a transaction event, it retrieves the user's active saving rules and computes the "save amount." For a round-up rule: `save = ceil(amount/spend_unit)*spend_unit - amount`. For percentage: `save = round(amount * percentage)`. It then initiates an internal transfer of `save` from the

user's spending wallet to a separate savings wallet (both under the same user). This transfer itself is a transaction (subject to balance checks). The service logs this save operation (date, amount, original tx reference).

- **Real-time Monitoring:** If the savings transfer fails (e.g. insufficient funds), the service can retry or alert the user. Users should have a view of all savings transfers in the app. Over time, these small transfers accumulate to help reach goals.

- **Group Savings (Chama):** Zanari plans a **chama/group savings** feature. Technically, this means supporting shared wallets. The engine can extend to group rules: e.g. each member's round-ups go into a common pot. Implement this by having group accounts in the database: when a group saving event triggers, the amount is credited to the group's wallet and the user's "debt" is noted. A smart contract (or just group ledger) can automate disbursement rules.

This approach ensures **immediate execution** of savings on each spend. It scales horizontally (many consumers, each spend just generates another event). It aligns with how apps like Acorns operate. Importantly, all saving transactions are on-platform and atomic. The one external dependency is the funding API (to debit user's actual funding source), but once the spend is captured, the rest is internal.

Example: User spends KSh 247 on food. Savings rule = "round to 50". The engine computes ceil(247/50)*50 - 247 = 3 KSh. It then debits 3 KSh from the user's* funding wallet* (which was debited by 247 anyway, so effectively debits 250 and re-credits 3 into savings) and credits 3 KSh to the user's savings balance. A small internal record "Saved 3 KSh from Tx #XYZ" is created.

8. Implementation Roadmap

Phase 1 – MVP (0–6 months):
- **Core Features:** Launch the basic wallet, user onboarding, and savings. Implement M-PESA integration first (largest market share) using Daraja in sandbox, then production. Set up Airtel Money integration if feasible early (it may be secondary). Allow wallet funding by mobile money and debit card (via partner). For debit card, start with *virtual cards only*. Use a card API (like Miden or Sudo) to issue virtual cards on registration. Integrate a payment gateway (e.g. Flutterwave or local PSP) if needed to accept card funding into the wallet.
- **Savings Engine:** Build simple save rules (e.g. fixed-percentage). Use serverless or single-node microservices to handle triggers. Store transactions in a SQL database.
- **Platform:** Deploy on a cloud quick-start (Firebase/Supabase with serverless functions) for speed, or use managed Kubernetes if team is ready.
- **Compliance:** Implement basic security (HTTPS, encryption), user KYC (upload ID), and DPA consent flows. We will not yet obtain any licenses but rely on Safaricom/Airtel for money flow.
- **Testing:** Thoroughly test payment flows (sandbox transactions). Prepare basic monitoring/alerts.

Phase 2 – Advanced Features (6–12 months):
- **Group Savings & Goals:** Add "chama" groups – shared saving goals with multiple members. Build UI and logic for group contributions and distribution (could use multi-signature or round-robin payouts).
- **Extended Integrations:** Activate T-Kash via aggregator or direct if possible. Consider adding bank deposit/withdraw (e.g. interbank transfers if Open Banking APIs exist, or pay bills to banks).
- **Enhanced Security/Compliance:** Move to a hardened containerized setup with full PCI compliance audit (if card usage increases). Possibly engage a security firm for pen-tests.

- **Offline Capabilities:** Develop USSD or SMS commands for basic functions (balance check, quick save, withdraw) so users with poor data can still interact. Also improve app for flaky networks (caching, retries).
- **Analytics and Personalization:** Deploy the analytics pipeline. Start showing dashboards (to admins) and personalized recommendations to users (e.g. “You could save an extra 5% based on your spending”). Use ML for fraud detection.
- **Performance Tuning:** Optimize database (sharding/indexes), add caching for high-read APIs (e.g. 2FA codes, KYC checks).

Phase 3 – Scale & Innovation (12+ months):
- **Full Card Program:** Offer physical debit cards (partner with printer). Integrate with ATMs if possible. Enhance virtual card controls (set limits by merchant, time).
- **Regulatory Compliance:** If required, pursue appropriate licenses (e.g. Payment Service Provider) and ISO/PCI certifications.
- **Emerging Tech:** Explore blockchain for transparent group savings contracts (especially if the number of users is very high). Implement chatbots (WhatsApp integration) for friendly UX. Use AI assistants to coach savings habits.
- **Optimizations:** Scale to multiple regions (if law allows) for redundancy. Continuously refactor services for micro-optimizations. A/B test new savings features.

Risk Mitigation:
- **Dependency Risks:** By using established services (Safaricom/Airtel gateways, card APIs, cloud platforms), we reduce the risk of having to re-invent regulated components. As a backup, support multiple payment gateways (e.g. Interswitch + Flutterwave) so that if one fails, others can cover.
- **Security Risks:** Early investment in security (encryption, audits) prevents costly breaches.
- **Market/Technical Risks:** Rapid MVP allows us to gauge user uptake before heavy investment. Use agile sprints to pivot quickly based on user feedback.

Timeline Estimate: MVP in ~4–6 months (dev team working full-time), advanced features by 9–12 months, full scale thereafter. These are accelerable with more resources or partnerships.

9. Innovation & Secondary Considerations

Emerging Technologies:
- *AI/ML:* Applying machine learning to user data could yield personalized saving plans (“we recommend you reduce spending on X to reach your goal faster”). Fraud detection models can flag unusual activity. Natural language processing (chatbots) can answer user queries or coach saving.
- *Blockchain:* Not strictly needed, but one could pilot using a blockchain (e.g. a private Ethereum or Hyperledger chain) to record group contributions/withdrawals for transparency among group members. Smart contracts could automate payouts for chamas. However, given complexity and cost, this may be a long-term explore.
- *Voice & USSD:* To reach more users, integrate with USSD (*XYZ#) so users without smartphones can still check balance or save. A possible service could “save X amount” via USSD.*
- Biometrics:* Use device biometrics (fingerprint/face) for app login to enhance security.

Partnerships & Integration:
- *Banks:* For KYC, integrate with M-PESA or bank APIs to confirm phone/ID. For disbursements, consider partnerships with payroll or microfinance. Technical: use banking APIs (in Kenya, look at KCB Bank API

portal) for instant account-to-wallet transfers.
- *Fintech Ecosystem:* Potentially allow depositing Crypto (like via Bitnob) and convert to KES savings. Or integrate with peer-to-peer lending for group savings use. Each integration would use the partner's API (likely REST/JSON, possibly requiring OAuth or API keys).

Offline & Connectivity:
In areas of poor internet, rely on SMS/USSD as noted. The app should cache last known data and function in "offline mode" (queue savings requests locally) then sync on reconnect. Progressive Web App (PWA) could allow some offline use, but the user's preference was mobile apps. If PWA is considered, Service Workers can cache static pages and maybe show last balances, but cannot process transactions offline without backend sync.

Cross-Platform Strategy:
Choosing React Native covers both iOS and Android natively. A PWA alternative could reduce installation friction, but PWAs have limited access to hardware (no card reader or secure credential storage). Given the focus on cards and real-time notifications, React Native (or Flutter) is superior for UX. We may eventually add a web portal (React) for admin or user access on desktop.

Data Analytics Infrastructure:
As noted, build on a robust data warehouse. Use tools like Apache Kafka (for streaming), Apache Spark or AWS EMR for large data processing, and BI tools (Metabase, or PowerBI on Azure) for dashboards. This supports continuous improvement (e.g. tracking how many users hit their saving goals).

Security & Compliance Checklist:
Ensure code is tested against OWASP ASVS. Plan for quarterly security audits. Maintain a comprehensive **security compliance checklist**: PCI DSS controls (if applicable), DPA articles (consent logs, DPIA), CBK audit trails. Use a Risk Assessment Matrix to evaluate technical (SQLi, XSS), regulatory (license requirements), and partnership (single vendor failure) risks and mitigations.

Cost & Time-to-Market:
By avoiding building a bank or pursuing new licenses, and by using partner services (mobile money APIs, card APIs), we accelerate deployment. The main cost will be platform development and API transaction fees. A progressive rollout (M-PESA first, then Airtel, T-Kash) balances cost/benefit. Use of React Native and managed backend services (Firebase or AWS Lambda) shortens dev time for MVP [15].

Key References: Integration details from Safaricom's Daraja portal and Airtel's developer docs [1] [3]; Multi-wallet support via Interswitch [5]; Card-issuing APIs (Miden/Sudo) [6] [7]; Kenyan Data Protection Act and compliance guides [11] [10]; Fintech stack best practices (cloud-native, Node/Go, containers) [8] [14]; and savings automation examples [12]. These inform a comprehensive, feasible architecture for Zanari in Kenya's fintech environment.

---

[1] Safaricom Daraja API: Authorization API Guide for Access Tokens - DEV Community
https://dev.to/msnmongare/safaricom-daraja-api-authorization-api-guide-for-access-tokens-2kg1

2 Demystifying The M-Pesa API (Lipa Na M-Pesa Online Payment) | by Denis Juma | The Andela Way | Medium
https://medium.com/the-andela-way/demystifying-the-m-pesa-api-lipa-na-m-pesa-online-payment-a22d68a42d5e

3 4 How to integrate Airtel money API for payment collections and remittances | by Agisha Muhanzi | Medium
https://medium.com/@muhanzi/how-to-integrate-airtel-money-api-for-payment-collections-and-remittances-67d2202cd488

5 Mobile Money Payments
https://interswitch-docs.readme.io/docs/mobile-payments

6 10 Best Card API in Kenya: How to Launch a Virtual Card Program for Your Business
https://blog.miden.co/best-card-api-in-kenya

7 _$udo | Modern card issuing and payments for Africa
http://sudo.africa/

8 9 14 15 16 17 18 Best Fintech and Insurtech App Stacks in 2025 - Award Winning Full Stack Digital Service Transformation Company | INT Global
https://intglobal.com/blogs/best-fintech-and-insurtech-app-stacks-in-2025/

11 Policy change - Digital Policy Alert
https://digitalpolicyalert.org/change/13684

12 What are Round-Up Savings | Benefits and Examples
https://www.arthamsu.com/what-are-round-up-savings/

13 Bitnob | Transforming Global Financial Pathways
https://bitnob.com/blog/the-best-virtual-card-for-online-subscriptions-in-kenya