ChatGPT

OnlyKAS P2P Merchant Platform Roadmap

Introduction

OnlyKAS is a Kaspa-native payment platform aiming to offer **Stripe-like functionality in a peer-to-peer (P2P) manner**. It enables merchants to accept instant cryptocurrency payments with support for **monthly subscription billing** and serves both brick-and-mortar **local stores** and **online web shops**. Unlike traditional processors, OnlyKAS operates **entirely on Kaspa's Layer-1** blockchain (no centralized servers or Layer-2 networks), ensuring merchants and customers transact directly. The Kaspa network's high throughput and fast confirmation times (10 blocks per second, with transactions fully confirmed within about one second [1]) make it ideal for real-time payments. The goal is to leverage Kaspa's speed and security to deliver near-instant settlement for everyday transactions [2], while providing features like recurring billing and dispute mediation in a decentralized fashion.

This roadmap outlines a plan to deliver an **MVP in 1 month** and a **full production rollout in ~3 months**, aligning closely with Kaspa's architecture and the **Kdapp** framework (Kaspa Decentralized Application framework). All development will be done in Rust, building on the **rusty-kaspa** codebase for seamless integration with Kaspa nodes and protocols. The project is community-driven and "cypherpunk" in spirit – merchants **self-host** their payment nodes, and no app-store dependencies are required for mobile usage (e.g. mobile apps can be sideloaded or run as web apps, avoiding centralized app stores). The result will be a self-sovereign payment network called **onlyKAS**, where participants transact freely and disputes are handled by a decentralized network of guardians rather than a traditional intermediary.

Project Goals and Key Features

- **Instant Payments on Kaspa L1:** Utilize Kaspa's blockDAG to achieve fast, direct payments with **1-second confirmations** and finality in ~10 seconds [2]. Customers pay merchants directly in KAS, with **no custodial middleman**.
- **Monthly Billing (Subscriptions):** Support recurring payments (e.g. monthly subscriptions). The system will allow merchants to bill customers on a schedule, either through customer-approved recurring charges or via monthly invoice reminders.
- **Local & Web Store Integration:** Provide solutions for both physical point-of-sale (POS) and online checkout:
- *Local Stores:* a lightweight POS application (e.g. a tablet/phone interface or terminal screen) for in-person payments (QR code scanning, etc.).
- *Web Stores:* APIs or plugins to integrate OnlyKAS payments into e-commerce websites, allowing customers to pay with KAS at checkout.
- **Mobile-Friendly, Self-Sovereign Setup:** Ensure the platform can run on mobile or small devices without relying on Google Play/App Store – for example, a Progressive Web App or direct APK install for Android, aligning with a cypherpunk ethos. Merchants and users retain control of their keys and can run the software on their own hardware.
- **Decentralized Dispute Resolution:** Implement a **guardian network** (distributed "watchtowers") to handle payment disputes or contract breaches. These guardians are independent nodes (part of the

OnlyKAS network) that can arbitrate and enforce outcomes (e.g. releasing escrowed funds if conditions are met or refunding a buyer in a proven dispute). This brings a chargeback-like safety net in a decentralized way.

- **Kaspa & Kdapp Alignment:** Design the system in accordance with Kaspa's architecture and the Kdapp framework:
- Use **Kaspa transactions** as the medium for carrying payment instructions and state updates (all logic anchored on the L1 chain, not a separate sidechain).
- Leverage Kdapp's **Episode** model for interactive sessions between merchant and customer, and its high-frequency transaction handling to track payments and subscriptions in real-time [3].
- Integrate with Kaspa nodes via wRPC (WebSocket RPC) for network communication, using the **rusty-kaspa** libraries for transaction construction, signing, and address management.

By meeting these goals, OnlyKAS will empower merchants to accept crypto with the ease of Stripe (automated invoicing, recurring billing, web integrations), while preserving the decentralization and speed of Kaspa.

Architecture Overview (Kaspa + Kdapp)

OnlyKAS is built on the Kdapp framework, which is designed for **interactive, high-frequency dApps on Kaspa** [3]. In this model, an application's logic runs off-chain but **uses on-chain Kaspa transactions to carry commands and state updates**. Key architectural components include:

- **Episodes (Payment Sessions):** Every payment or subscription can be modeled as a Kdapp **Episode**, which represents an interactive session between the merchant and customer. For a one-time purchase, an episode might span from invoice creation to payment confirmation. For a subscription, a long-running episode could manage the state across billing cycles. The Episode trait will encapsulate the business logic (e.g. verifying payments, scheduling next bill, handling cancellations).
- **Engine and Proxy:** The Kdapp Engine runs the episodes and maintains their state, while the Proxy component listens to the Kaspa network (via wRPC) for relevant transactions [4]. OnlyKAS will define a unique transaction pattern or prefix for its episodes (using the Kdapp **Generator** to craft transactions with identifiable IDs). The Proxy filters the global Kaspa mempool/block stream for transactions that match OnlyKAS's pattern (for example, an OP_RETURN or specific prefix in the transaction ID) and forwards those to the Engine [5]. This way, when a customer makes a payment or a scheduled event triggers, it's recorded on-chain and the Engine immediately picks it up.
- **Kaspa Transaction Mechanics:** Payments are performed with regular Kaspa transactions (sending KAS to the merchant). For simple instant sales, this is a direct transfer to the merchant's address. For subscriptions or escrow scenarios, OnlyKAS will leverage Kaspa's UTXO script capabilities:
- *Multi-Signature Escrow:* To enable dispute resolution, a payment can be locked in a 2-of-3 multisig output (customer, merchant, and a guardian key). The Engine (and guardians) coordinate to sign the spending transaction depending on outcomes (e.g. merchant gets funds on successful delivery, or customer gets refund if a dispute is resolved in their favor). This mimics an escrow/chargeback process without a centralized party.
- *Time-Locks for Subscriptions:* For recurring billing, we may use absolute time-locks (CSV/CLTV) in scripts or simply rely on off-chain scheduling. One approach for automation is having the customer pre-sign transactions for future payments (or authorize a smart contract-like script) that the merchant can trigger each month, with the ability for the customer to revoke before execution.

Guardians/watchtowers would oversee that these pre-signed transactions are not published prematurely or fraudulently.

- **Guardian Nodes (Watchtowers):** Guardians are essentially **Kdapp nodes running in watchtower mode** across the OnlyKAS network. They observe the Kaspa blockchain for protocol violations or for the need to step in. For instance, if a malicious party tries to post an outdated state or steal escrow funds, guardians detect it and broadcast the appropriate counter-transaction (using pre-signed justice transactions or their own signatures as needed) to enforce the correct outcome. In practice, guardians could be community-run servers that merchants and customers opt into for trust. They form the backbone of dispute resolution – rather than trusting a credit card company or Stripe to mediate, the protocol and guardians handle it in a distributed way. (Because Kaspa's base layer doesn't natively support smart contracts, this functionality is achieved through off-chain coordination and multi-sig scripts on-chain, managed by the Kdapp logic.)
- **Rusty-Kaspa Integration:** The implementation will align with **rusty-kaspa's architecture** by using its APIs for transaction building, address derivation, and consensus rules. The Kdapp engine itself relies on Kaspa's consensus via wRPC; e.g., OnlyKAS can connect to a local `kaspad` node or a public Kaspa node to send/receive transactions [6]. All consensus-critical code (like transaction validation, script execution) is handled by Kaspa's proven codebase. OnlyKAS adds a layer on top for coordinating sequences of transactions and enforcing business rules (e.g., "payment due monthly"). This layered approach means we **don't modify Kaspa's core**, but rather build atop it – ensuring full compatibility with the network and benefiting from Kaspa's security and performance out of the box.

Overall, the architecture ensures that OnlyKAS acts as a **non-custodial facilitator**: it helps merchants and customers communicate and transact through the Kaspa blockchain. Kdapp provides the fast off-chain engine to react to on-chain events, effectively letting us run **real-time payment logic on a high-throughput PoW network** (Kaspa's 10 BPS block rate) [7]. This is a novel approach that treats L1 not just as a settlement layer but as a messaging bus for interactive protocols. The trade-off is some complexity in state management (since state is not stored in a global smart contract, the Engine must track it and handle re-orgs via rollback if needed [8]), but this is acceptable in exchange for decentralization and speed. We will mitigate risks by rigorous testing and possibly persisting important state to disk as a backup (in case an engine restarts, it can recompute episode state from the chain history).

With the architecture defined, we now break down the development roadmap into two main phases:

Phase 1: MVP (Month 1)

Timeline: ~4 weeks for initial prototype.
Goal: Deliver a minimal viable product focusing on core payment functionality and a basic subscription mechanism, proving that P2P Kaspa payments can work for merchants. The MVP will be tested on Kaspa testnet and prepared for an early demo.

Key deliverables in MVP:

- **Core Payment Transactions:** Implement the ability for a merchant to create a payment request and for a customer to pay it in KAS. For MVP, this will be a **direct one-time payment** flow. Example: a merchant runs the OnlyKAS app to generate an invoice (amount, description) which yields a QR code or "payment key". The customer uses the OnlyKAS client (or a standard Kaspa wallet in simple cases)

to send the payment. The system confirms the transaction on-chain and notifies the merchant instantly. This requires:

- Generating Kaspa addresses or episode keys for each invoice.
- Using the Kdapp **Generator** to produce a transaction (or transaction template) with a recognizable ID/pattern [5]. In MVP, we might simplify and just watch a specific address for incoming payments if pattern-matching is not fully implemented yet.
- The Proxy listening to the Kaspa network and catching the payment transaction in real-time, triggering an acknowledgment to the merchant's UI.
- **Basic Recurring Billing Framework:** Lay the groundwork for subscriptions. In MVP, this could be as simple as allowing a merchant to mark an invoice as "recurring monthly" and storing that info in the Engine. When the due date arrives (e.g. every 30 days), the system should generate a new invoice or alert for the customer. MVP will not fully automate the payment each month, but it will demonstrate the cycle:
- The Engine keeps track of subscription state (next due date, whether last payment received).
- Perhaps send a notification or print a message when a periodic payment is due, to be handled manually at first.
- (Stretch goal within MVP: implement a **dummy auto-payment** on testnet – e.g., have the customer pre-sign the next transaction and let the Engine auto-broadcast it at due time, to prove the concept of automated billing.)
- **Kaspa Network Integration:** Ensure the application can connect to Kaspa. Out-of-the-box, we'll allow connection to **Kaspa public nodes (PNN)** or a local `kaspad`. For simplicity, MVP will default to testnet settings. We'll use Kaspa's wRPC via the Kdapp Proxy to subscribe to mempool events and new blocks [6], so that the merchant doesn't need to poll for payments – the app will react as soon as the transaction is seen. Transactions will be signed and broadcast through the same connection. Using Kaspa's testnet (and faucet for funds) will allow rapid iteration without risk.
- **Local Store POS Prototype:** Develop a minimal **Point-of-Sale interface** for in-person payments. This can be a command-line or terminal UI in MVP, but it should allow a merchant to input an amount and then display a QR code representing the payment request (e.g., a URI with Kaspa address and amount). The customer can scan this with their wallet to pay. Once payment is detected, the terminal will show confirmation (e.g., "Payment received: 50 KAS from Alice"). Even at MVP stage, thanks to Kaspa's speed, the confirmation can be nearly instant (within 1–2 seconds typically) [1], giving a cash-like experience at checkout.
- **Web Checkout Demo:** Provide a simple example of web integration – for instance, a dummy **checkout page** (HTML/JavaScript) that interacts with the OnlyKAS backend. In MVP this could be a basic webserver in the OnlyKAS app that serves an invoice page at `http://localhost:8000/pay?amount=X`. When accessed, it generates a new payment episode and displays the QR code or payment link. Once the Kaspa transaction is received, the page could automatically update to "Payment complete". This will prove that online stores can use the system (the full plugin/API will be expanded in Phase 2).
- **Security and Key Management:** Implement basic key handling. The merchant will need a Kaspa keypair to receive funds (or a new key per invoice). In MVP, we'll likely use an **embedded wallet** approach (similar to the Kdapp tic-tac-toe example, which generates a keypair on first run [9]). The app will either prompt the merchant to input their Kaspa private key (if they have one) or generate a new one and show them the address (with instructions to fund it for testing). For now, keys can be stored in a local config file or environment variable. Emphasize to MVP testers that this is experimental and not to use real large funds yet. Basic encryption of keys (or integration with system

keystores) can be planned later; for MVP, simple file storage is acceptable but we will caution users about security.

- **Self-Serve Deployment (Onboarding):** Document a straightforward process for a merchant to get started on their own. Since this is P2P, the merchant is responsible for running the software. We will provide a **README or guide** covering: installing Rust (if needed) and building the binary, running the OnlyKAS node, generating an address, and accepting a first payment. This guide will double as a testing checklist for the MVP. If possible in this timeframe, create a helper script or interactive prompt in the app to do initial setup (e.g., "Press 1 to configure your store" which walks them through setting name, getting an address, etc.). The aim is that a moderately technical user can follow instructions and have a basic Kaspa payment terminal running in under an hour.
- **Testing & Demo:** Before moving to Phase 2, we will rigorously test the MVP:
- Simulate a few payment scenarios on Kaspa testnet: normal payment, payment with network latency, small reorg (ensure our Kdapp engine rollback works – e.g., if a payment tx appears then gets orphaned, the system should handle it gracefully and wait for the next block [8]).
- Test the subscription cycle logic with shorter intervals (e.g., set "monthly" to every 5 minutes on testnet to quickly see it generate follow-up invoices).
- Document any limitations or bugs discovered and decide which are critical to fix now vs. later.
- Prepare a live demo (or recording) showing a merchant device and a customer wallet performing a transaction via OnlyKAS to showcase to stakeholders.

MVP Outcome: After one month, we expect to have a working prototype that can process instant Kaspa payments in a real P2P fashion. A merchant will be able to run their own instance and accept payments both in-person and via a simple web demo. Some features will be rudimentary (e.g., manual renewal of subscriptions, basic UI), and robust dispute resolution won't yet be in place, but the core concept – **"Stripe-like payments over Kaspa"** – will be proven. This sets the stage for expanding functionality and polishing the system in Phase 2.

Phase 2: Full Rollout (Months 2–3)

Timeline: ~8–12 weeks (following MVP) for feature-complete product and deployment on Kaspa mainnet.
Goal: Expand the MVP into a full production-ready platform, with all key Stripe-like features implemented: automated subscriptions, robust dispute/escrow support, polished user experience for both local and web contexts, and security hardening. By the end of Phase 2 (around the 3-month mark), OnlyKAS should be ready for real-world use on Kaspa's mainnet by early adopters.

Key deliverables in full rollout:

- **Automated Recurring Payments:** Build on the MVP subscription framework to support true automated billing. There are a couple of approaches, and we may implement a combination:
- *On-Chain Scheduling:* Utilize block height or timestamps for scheduling future payments. For example, when a customer subscribes, the system could create a series of post-dated transactions (or a single transaction with a time-lock that can be activated monthly by the merchant). Kaspa doesn't have internal scheduling, so the Engine will handle triggering at the right time, but we can encode the schedule into a script if needed (e.g., outputs locked until certain block heights).
- *Pre-Signed Transactions:* A user could pre-authorize payments for the next N months. Technically, they would sign N transactions (one for each future month's payment) and give these to the Engine (or a guardian). The Engine will store them and broadcast each at the right time. The user retains the

ability to cancel by spending the funds or revoking via a conflicting transaction if they decide to unsubscribe, and guardians would ensure the merchant can't grab funds early.

- *Off-chain Invoicing with Reminders:* As a more user-friendly alternative, the system will also support simply sending the customer a notification (email, message, or wallet notification if using an OnlyKAS-aware wallet) each billing period, and the customer manually approves the payment. This is akin to invoice billing (as Stripe offers in subscriptions with invoices). It's simpler and avoids locking funds upfront, at the cost of requiring user action each time. We will implement this notification system (perhaps via email or a lightweight mobile app alert) so that even if fully automated payment is not set up, the subscription can continue smoothly with the customer's involvement.
- *Subscription State Machine:* Enhance the Episode logic to handle states like **Active**, **Past Due**, **Cancelled**. For instance, if a payment is missed, the episode can mark it as Past Due and optionally notify guardians or impose penalties (maybe notify the merchant to pause service). This logic ensures that both parties and any observers have a consistent view of the subscription status.
- **Guardian Network & Escrow Dispute Resolution:** Introduce the OnlyKAS **guardian network** into the live system:
- Recruit or encourage community members (or possibly a set of foundation-run nodes initially) to run guardian nodes that will participate in dispute arbitration. Document how one can run a guardian (likely running the OnlyKAS software in a special "watchtower mode" that doesn't require acting as merchant or customer, just observing and responding).
- Implement the **escrow payment option** for transactions. In the MVP, payments were direct. Now, for higher-value transactions or unfamiliar counterparties, a merchant can opt into an escrow mode: the payment goes into a 2-of-3 multisig (merchant, customer, guardian). By default, if everything is good, merchant and customer (or merchant + guardian) will finalize the tx to merchant after, say, the customer confirms delivery. If there's a dispute, the guardians can decide to sign with one party to refund or release funds based on evidence. This feature requires:
    - Multi-sig address generation and management in the app (possibly using rust-kaspa's script capabilities).
    - A protocol for **dispute submission**: e.g., if customer claims non-delivery, they trigger a "dispute" command in the Episode. Guardians then wait a certain period for evidence (which might be off-chain, so this part may be more of a social/trusted process – maybe via a forum or attached data). After review, guardians either do nothing (letting merchant time-lock expire to get funds) or they co-sign a refund transaction to customer. The specifics of *how* guardians get evidence may be beyond the system (could be out-of-band), but our job is to provide the tools (multisig transactions, time-locks, and a way for guardians to intervene).
    - Ensure that guardians cannot themselves be malicious without collusion. Perhaps require multiple guardians to agree (multi-watchtower scenario) or allow merchants/customers to choose a set of guardians they trust. In Phase 2, a simpler approach might be designating a few neutral guardians for the beta rollout. As OnlyKAS grows, this can evolve into a decentralized reputation system.
    - Testing the dispute flow extensively: intentionally create a scenario (on testnet) where a dispute is invoked and ensure the guardian logic correctly resolves it (money goes to the right party) and that the losing party cannot cheat (thanks to Kaspa's immutability and the multi-sig requiring a guardian signature).
- The presence of guardians also helps enforce correct protocol usage. For example, in a subscription context, if a merchant tried to "pull" a payment earlier than scheduled (with a pre-signed tx), guardians would detect the violation (timestamp not reached) and refuse to sign or would alert the customer to revoke consent. In essence, guardians act as **referees** ensuring neither side violates the

agreed terms encoded in the OnlyKAS episode. This decentralized oversight is a key differentiator of OnlyKAS, giving users confidence despite the lack of a centralized authority.

- **Enhanced Merchant Experience (UX/UI):** Refine the interfaces for real-world use:
- **Merchant Dashboard:** Develop a simple graphical UI (possibly a web-based dashboard or a cross-platform app using a Rust GUI framework or Electron) for merchants to manage their OnlyKAS instance. This would allow viewing incoming payments, subscription statuses, and creating new payment requests with a form (instead of command-line). It should also show alerts (e.g., "Subscription X is due for renewal" or "Guardian action required for dispute Y"). In Phase 2 we may not build a full polished product like Stripe's dashboard, but we aim for a functional, user-friendly interface that lowers the technical barrier. If time is short, prioritize a **web UI** served by the OnlyKAS node itself (so the merchant can open a local webpage to manage their store).
- **Mobile POS App:** As many small merchants prefer smartphones or tablets, create a **mobile-friendly interface**. This could be done by ensuring the web dashboard is responsive for mobile screens, or by packaging it as a PWA (Progressive Web App) so it can be "installed" on a phone. Alternatively, provide an Android APK that basically loads the web UI in a WebView and has camera access for scanning QR codes. Importantly, this app will be distributed directly (via GitHub or IPFS) to avoid reliance on app stores, fitting the cypherpunk vibe. It will function like a "payment terminal in your pocket" for Kaspa.
- **Web Store Integration Tools:** Develop plugins or libraries to simplify integration with common e-commerce platforms. For instance:
    - A WooCommerce/WordPress plugin that communicates with a merchant's OnlyKAS node to create an invoice at checkout and verify payment before marking an order paid.
    - A JavaScript SDK that developers can add to custom websites. This SDK could open a payment popup or redirect to an OnlyKAS payment page, similar to how Stripe Checkout works, but all hosted by the merchant's own server. We might provide a reference implementation of a **"checkout widget"** that queries the OnlyKAS backend for a payment token and then listens via WebSocket for confirmation to update the UI.
    - Clear API documentation for developers: e.g., an HTTP endpoint on the OnlyKAS service like `POST /create_invoice` returning a payment ID and payment details (amount, address or QR, etc.), and maybe webhooks or callbacks for status. In Phase 2, we'll implement at least the API and one example integration (likely a simple plugin or script) to prove it works.
- **Performance and Scalability:** Optimize the system to handle a growing load:
- Kaspa can handle a high TPS, but our OnlyKAS node must efficiently filter and process relevant transactions. We will profile the Kdapp Engine with many concurrent episodes (e.g., simulate 100+ active invoices) to ensure it can keep up. If needed, implement indexing (the "Generator pattern" approach means we only listen for transactions that match a pattern, which is efficient [10], but we should confirm performance on mainnet conditions).
- Improve the rollback and state management in the Engine. For example, implement the **in-memory rollback cap** suggested in Kdapp docs (limit stored undo history to a reasonable number) [11] to prevent memory bloat if a merchant has thousands of transactions. Also, consider persisting episode states to disk so that if the service restarts, it can recover without rescanning entire chain (or provide a fast catch-up by scanning only recent blocks or relying on Kaspa's index if available).
- Ensure that multiple OnlyKAS instances (many merchants) can run without interference. Since each uses unique keys/patterns, this should be fine, but we might stress test running multiple on one machine or one network to simulate many merchants operating.
- **Security Enhancements:** With real money at stake on mainnet, fortify the security:

- **Key Security:** Upgrade key storage from MVP's plain storage. Options include integrating with hardware wallets (if Kaspa is supported via e.g. Ledger or a standard like BIP32 for derivations), or at least encrypting keys with a passphrase in the config. Ensure the software never transmits private keys and that sensitive actions (like guardian signing) require explicit permission if the key is hot. Potentially, allow merchants to operate in a watch-only mode where the OnlyKAS node doesn't hold the spend key but can still detect payments – the merchant would then manually move funds from their wallet as needed (this might be useful for those who want extra security, though it means no automated spends from that instance).
- **Code Audit & Testing:** Conduct thorough code reviews, possibly engage community or external auditors to review the critical crypto and consensus handling parts. Because we align with Kaspa's core libraries, risk is reduced on the consensus side, but our custom logic for subscriptions and disputes should be scrutinized for any loopholes (especially anything that could be exploited to steal funds or cheat the system). Write unit tests and integration tests for various scenarios (e.g., simulate a malicious actor trying to broadcast an old state or a false guardian signature).
- **Guardian Trust Model:** Define clearly how guardians are selected and trusted in this early stage. Since guardians could, in theory, collude, we might initially only use a limited set of known guardians (perhaps run by community-vetted members or the OnlyKAS developers) for the beta. Long-term, we want a decentralized set, but that might require reputation systems or staking which are beyond 3 months scope. For now, document the recommended guardian selection process (e.g., merchant and customer mutually agree on a guardian or two when initiating an escrow). Possibly implement a feature where the guardian's public key is embedded in the episode at start (so both parties know who will mediate).
- **Merchant Onboarding & Documentation:** By rollout, make it as easy as possible for an average user to start using OnlyKAS:
- Create a step-by-step **Merchant Onboarding Guide** with screenshots (if GUI) or example commands. This should cover installation (maybe providing pre-built binaries for common platforms to avoid requiring Rust toolchain for non-developers), initial configuration (creating or inputting a Kaspa key, setting up connection to a Kaspa node), and basic usage (creating an invoice, receiving payment, checking status).
- Provide **FAQ and Troubleshooting** docs. Anticipate common issues (network connectivity, firewall for wRPC, losing a key, etc.) and address them. Also explain concepts like guardians and recommendations on when to use escrow vs. direct pay. Since this is new, educating users on the trust trade-offs is important (e.g., "If you don't know the customer, consider using a guardian escrow to protect both parties").
- If possible, implement an **interactive onboarding mode** in the app: for example, on first run, detect no config and walk the user through generating a key and connecting to Kaspa. This could significantly improve user experience and reduce errors in setup.

- Community support: set up a channel (perhaps a Discord or Telegram) where merchants can ask questions during the beta. This isn't a code deliverable per se, but part of rollout readiness to ensure any snags can be addressed quickly.

- **Mainnet Deployment and Testing:** In the latter half of Phase 2, transition from testnet to **Kaspa mainnet**:

- Before enabling real payments, test a small-scale mainnet pilot with a friendly merchant and customer to make sure everything works with real KAS (confirm fee handling, etc.). Kaspa's mainnet

operates similarly but we'll double-check performance and any differences in wRPC endpoints or network quirks.
- Implement any needed adjustments for mainnet scale (for instance, if our transaction pattern was fine on testnet but needs tweaking for mainnet volume to avoid collisions or to ensure uniqueness).
- Once confident, release a **v1.0** of OnlyKAS Merchant. Announce it to the Kaspa community and encourage trial usage. Monitor the network and feedback closely in the initial days. Have guardians on standby to intervene in any test disputes to prove that aspect works live.
- Collect metrics: how fast payments confirm in practice, any reorg events handled, how users find the experience. Use this data to further refine (likely beyond the 3-month mark as ongoing improvement).

By the end of Phase 2 (around 3 months in), OnlyKAS will have evolved from a prototype into a **feature-rich P2P payment solution**. It will allow a merchant to operate almost like they would with Stripe or PayPal – generating payment links, handling subscriptions, etc. – but *without any centralized platform taking fees or custody*. All transactions are settled directly on Kaspa, meaning merchants receive funds in their own wallet instantly. The addition of the guardian-based escrow system provides an answer to the lack of chargebacks in crypto: when needed, transactions can be made reversible/conditional with mutual consent and third-party oversight, all encoded in the open protocol.

Future Outlook and Conclusion

The 3-month roadmap delivers a functional Stripe-like service over Kaspa, but the journey doesn't end there. Looking beyond the initial rollout, we envision: improved scalability (leveraging Kaspa's roadmap to 100 BPS in the future [12]), more user-friendly mobile apps for customers, possible integration with hardware terminals for retail, a reputation system for guardians and merchants, and AI-assisted "vibe coding" to auto-generate merchant workflows (echoing the long-term vision of Kdapp's creators [13]).

In conclusion, this roadmap provides a detailed path to achieving a **Kaspa-aligned P2P merchant system** that brings together the best of both worlds: the convenience and business features of Stripe, and the decentralization and speed of the Kaspa blockchain. With an MVP in one month and a full rollout in three, OnlyKAS is on track to demonstrate that truly decentralized payment networks for commerce are not only possible, but imminent. The project stays true to Kaspa's architecture and the Kdapp framework throughout, ensuring compatibility and leveraging the unique strengths of Kaspa's blockDAG (high throughput, fast confirmation) [1] [7]. By empowering merchants to self-host and by utilizing community guardians for trust, OnlyKAS aims to create a **self-sustaining, peer-to-peer "payment net"** – one that embodies the ethos of crypto by removing middlemen and giving control back to users, without sacrificing the reliability and features merchants need.

Sources:

- Kaspa BlockDAG & Speed – *Kaspa Features*: *"Kaspa's blockDAG network generates 10 blocks every second… Combined with fully confirmed transactions in 1 second, this makes Kaspa ideal for everyday transactions."* [1]; *"each Kaspa Transaction confirmed to the network in one second, and each transaction fully finalized in 10 seconds on average."* [2]
- Kdapp Framework – *Michael Sutton's Kdapp README*: *"Kdapp provides the infrastructure to build semi-native, interactive, and time-sensitive decentralized applications (k-dApps) on Kaspa… leverage Kaspa's unique 10 blocks-per-second capability."* [3]; *Architecture: "Generator: crafts Kaspa transactions with*

specially formatted payloads… matching a predefined pattern… Proxy: wRPC client that listens… for transactions matching the pattern… forwards to the core engine.” [5] ; *Engine: “interprets incoming commands, validates signatures, updates episode state, and maintains a stack of rollback objects to handle Kaspa DAG re-organizations.”* [14] .

---

[1] [2] [12] Features - Kaspa
https://kaspa.org/features/

[3] [4] [5] [6] [7] [8] [9] [10] [11] [13] [14] GitHub - michaelsutton/kdapp
https://github.com/michaelsutton/kdapp