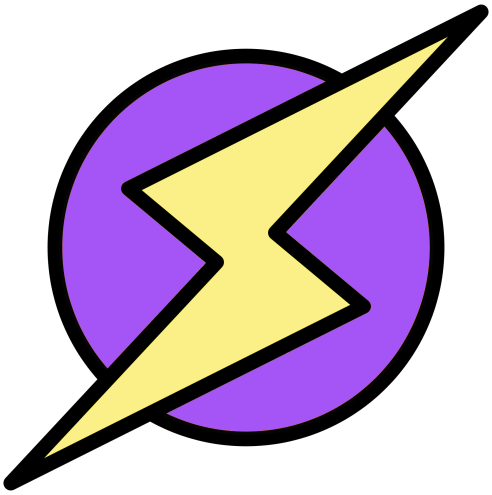
Gautam Manchandani
Payment Escrow and Processing



SOB 2025 Proposal

Shopstr Escrow & Dispute Resolver: Streamlined Payment Locking and Automated Order Processing for Secure Marketplace Transactions

TABLE OF CONTENTS

Cover 1
Table of Contents 2
Synopsis 3
Introduction 4
- Personal Details 4
- Open-Source Contributions 4

Why SOB and this Project ? 5
Overview of Current Shopstr workflow 5-8
Approach/Implementation 9-18
- P2PK Time-Locked Cashu Tokens 9
- Automated Processor for Payment & Shipping 12
- UI To handle Disputes between Seller & Buyer 14
- Add HODL Invoices 17

Timeline & Deliverables 19
Post SoB & Prior contributions 20

Synopsis

The goal of this project is to enhance Shopstr's payment processing and dispute handling system by automating fund release and introducing dispute resolution features. Right now, Shopstr's payments and order confirmations happen manually. This project will build on Shopstr's existing Cashu proof-based escrow by adding true P2PK pubkey-locking (with HODL invoices as another option so that funds remain locked until both buyer and merchant confirm that order conditions have been met. Additionally, we plan to develop a new user interface and backend logic to enable raising and handling disputes—initially by letting Shopstr act as the sole moderator—to secure both parties and streamline communication. In short, this project will reduce risks associated with pseudonymous transactions, ensure sellers do not need to be online at purchase time, and provide a clear and trustworthy process to recover funds in case of problems.

Benefits to the Users

- **Enhanced Payment Security:** Funds are locked in escrow until both the buyer and seller confirm that the order is fulfilled. This minimizes the risk of fraud because money isn't released until conditions are met.

- **Faster, Automated Order Processing:** With an automated system handling payment confirmations and shipping messages, sellers don't need to be online immediately. This makes the process faster and smoother for buyers and sellers alike.

- **Transparent Dispute Resolution:** A clear, easy-to-use interface for raising and tracking disputes gives users peace of mind. If something goes wrong, users have a straightforward way to escalate the issue and recover funds when necessary.

Shopstr Escrow & Dispute Resolver: Streamlined Payment Locking and Automated Order Processing for Secure Marketplace Transactions

SOB'25

Name: Gautam Manchandani
Major: Computer Science and Engineering
Degree: Bachelor of Science (BSc Hons.)
Year: Sophomore
University: Birla Institute of Technology and Science, Pilani (Rajasthan, India)
Github Handle: @gautambytes
Email: gautammanchandani0@gmail.com
Phone: (+91-9630405239)
Resume: Link
Portfolio Website: Link
Timezone: Indian Standard Time (UTC +5:30)

About me

I'm a full-stack developer with experience building AI-powered applications, including a medical chatbot and an AI-based education platform. My passion for open source and decentralized solutions extends to Bitcoin-related projects—Bitcoin Themed Time Capsule (a Solidity contract for time-locked messages) and Bitcoin Timestamp Verifier (a React-based tool to verify documents on-chain). These diverse projects reflect my drive to leverage technology in ways that empower end users and broaden access to innovation.

You may know me better as GautamBytes on github!

My Open-Source Contributions

I started contributing to open source during Hacktoberfest 2024, merging over 10 PRs across various repositories. For the past two months, I've focused on Learning Equality, where I've added more than 10 merged PRs in areas ranging from UI fixes to code refactoring. Through regular Slack discussions and code reviews, I've deepened my understanding of community-driven development and gained insights into effectively collaborating within established projects. [Merged PR's Till date]

Why SOB and particularly the Payment Escrow Project ?

I first heard about SOB in my first year of college, but back then I was still getting comfortable with programming, especially with new languages like Rust and TypeScript. After winning a few hackathons and contributing to open source projects like Learning Equality and Hacktoberfest'24, I gained the confidence to dive into SOB. I'm excited about this project because I see a chance to really improve Shopstr by automating payment escrow using Cashu's P2PK locked tokens (with HODL invoices as an option) and building a clear, user-friendly dispute resolution UI. This work will help reduce friction between buyers and sellers while strengthening trust in pseudonymous transactions, and I believe my experience and most importantly honest interest makes me a great fit to contribute meaningfully.

Brief Overview of the Current Shopstr Workflow :

1. Cashu Token Payment (Standard Tokens):

Shopstr's current flow includes functions to "publish" payment events that generate standard Cashu tokens. The process creates a Cashu proof event, but unlike P2PK tokens, these tokens can be claimed by anyone who has access to them - they aren't locked to a specific recipient's public key. Once the proof event is signed and published, the tokens remain in the system until someone manually claims them.

For example, the function publishProofEvent demonstrates how the code encrypts, signs, and publishes a Cashu proof event to the network, but without any pubkey-based locking mechanism.

```
export async function publishProofEvent(
  nostr: NostrManager,
  signer: NostrSigner,
  mint: string,
  proofs: Proof[],
  direction: "in" | "out",
  amount: string,
  deletedEventsArray?: string[]
) {
  try {
    const { relays, writeRelays } = getLocalStorageData();
    const allWriteRelays = withBlastr([...relays, ...writeRelays]);
    const userPubkey = await signer?.getPubKey?.();
```

2. Lightning Invoice Option:

Shopstr provides a mechanism for generating a Lightning invoice as an alternative payment option. When a buyer initiates a payment, the system generates an invoice that specifies the amount due. While the invoice informs the buyer how much to pay and is created programmatically, the actual settlement remains intertwined with the Cashu token flow (i.e. the invoice is used to calculate payment but the escrow is managed with Cashu tokens).

Below code calls the Lightning API (via methods like requestInvoice) to generate an invoice for a given amount. This invoice informs the buyer of their payment obligation, even though the Cashu escrow mechanism still governs fund locking.

```
if (wallet) {
  await wallet.loadMint();
  await ln.fetch();
  const invoice = await ln.requestInvoice({ satoshi: newAmount });
  const invoicePaymentRequest = invoice.paymentRequest;
  const meltQuote = await wallet.createMeltQuote(invoicePaymentRequest);
  if (meltQuote) {
    const meltQuoteTotal = meltQuote.amount + meltQuote.fee_reserve;
    const { keep, send } = await wallet.send(meltQuoteTotal, proofs, {
      includeFees: true,
    });
```

3. Dispute Resolution:

Presently, there is no fully implemented dispute resolution workflow. Some parts of the UI hint at mechanisms for sending inquiries and perhaps a placeholder for "Leave a Review" or "Shipping Info" (which might later tie into dispute resolution). However, a dedicated "Raise Dispute" button or module specifically for resolving disputes is not yet built.

```
<div className="flex items-center justify-between border-t p-4">
  <Button
    className={SHOPSTRBUTTONCLASSNAMES}
    onClick={handleToggleReviewModal}
  >
    Leave a Review
  </Button>
</div>
```

```
<div className="flex items-center justify-between border-t p-4">
  <Button
    className={SHOPSTRBUTTONCLASSNAMES}
    onClick={handleToggleShippingModal}
  >
    Send Shipping Info
  </Button>
</div>
```

(Above code snippets are from chat-panel.tsx, there are buttons for actions such as "Send Shipping Info" and "Leave a Review." These currently trigger manual actions—serving as a basis to later extend the functionality into a dispute resolution workflow.)

4. **Confirmation & Shipping Messages:**

Once a purchase is made, the seller manually confirms the order and shipping through the UI. The "Claim" button (in claim-button.tsx) contains logic that, upon clicking, processes a redemption. Part of that process involves generating a Lightning invoice, processing a "melt" quote via the Cashu wallet, and ultimately sending a message (using a "gift-wrapped" Nostr event) to notify the buyer.

```
const redeem = async () => {
  setOpenClaimTypeModal(false);
  setOpenRedemptionModal(false);
  setIsRedeeming(true);
  const newAmount = Math.floor(tokenAmount * 0.98 - 2);
  const ln = new LightningAddress(lnurl);
  try {
    if (wallet) {
      await wallet.loadMint();
      await ln.fetch();
      const invoice = await ln.requestInvoice({ satoshi: newAmount });
      const invoicePaymentRequest = invoice.paymentRequest;
      const meltQuote = await wallet.createMeltQuote(invoicePaymentRequest);
      if (meltQuote) {
        const meltQuoteTotal = meltQuote.amount + meltQuote.fee_reserve;
        const { keep, send } = await wallet.send(meltQuoteTotal, proofs, {
          includeFees: true,
        });
```

Below modal informs the user (buyer) that the purchase went through and that the seller has been notified via a DM. The actual creation and publication of these events occur in the Claim button's redeem function (as shown above), which is manually triggered at the moment.

```
<ModalContent>
  <ModalHeader className="flex items-center justify-center text-light-text dark:text-dark-text">
    <CheckCircleIcon className="h-6 w-6 text-green-500" />
    <div className="ml-2">Purchase successful!</div>
  </ModalHeader>
  <ModalBody className="flex flex-col overflow-hidden text-light-text dark:text-dark-text">
    <div className="flex items-center justify-center">
      The seller will receive a DM with your order details.
    </div>
  </ModalBody>
</ModalContent>
```

Approach / Implementation:

For explaining the approach , I will be dividing expected outcomes in 4 different parts as below:

– Implementing P2PK locked Cashu tokens and adding time-locks to hold funds until each party confirms the necessary spending conditions.

– Adding HODL invoices as another optional feature to hold funds.

– A new UI to allow for the opening of disputes to be handled between the merchant and seller and an optional third-party.

– An automated processor acting on behalf of the merchant to confirm payments and process confirmation and shipping messages.

1. P2PK Time-Locked Cashu Tokens :

Below are the steps explaining my approach to implement this -

Step 1: Extend the Minting Flow to Attach a Time-Lock

Right now , in our mint-button.tsx file, when a buyer makes a payment (after paying a Lightning invoice), we call the wallet's minting functions (like wallet.mintProofs(...)). The proofs received are then passed to a function such as publishProofEvent(...) (defined in nostr-helper-functions.ts) that creates a "proof event." These proofs are stored (often in localStorage) and later manually redeemed by the seller through the "Claim" button in claim-button.tsx.

For example, the current code publishes a proof event as follows:

```
await publishProofEvent(
  nostr!,
  signer!,
  mints[0]!,
  proofs,
  "in",
  numSats.toString()
);
// The proofs (and their associated token) are then stored locally.
```

- **Calculate and Attach a Locktime:**
  When proofs are received, we will immediately calculate an unlock time. For example, if we want the tokens to be locked for one hour, we compute:

```
const lockDelaySeconds = 3600; // 1 hour delay
const locktime = Math.floor(Date.now() / 1000) + lockDelaySeconds;
```

- **Stamp Each Proof:**
  We then extend each proof by adding a new property (e.g., locktime). In the context of our existing code, this means modifying the proofs array right after they're generated:

```
while (true) {
  try {
    const proofs = await wallet.mintProofs(numSats, hash);

    if (proofs) {
      const proofArray = [...tokens, ...proofs];
      localStorage.setItem("tokens", JSON.stringify(proofArray));
      localStorage.setItem(
        "history",
        JSON.stringify([
```

- **Token Encoding:**
  When we call our token encoding function (for instance, getEncodedToken({ mint: tokenMint , proofs: changeProofs}) in claim-button.tsx), the new locktime property is

- embedded into the token. This way, later in the redemption process the system can read and enforce the time lock.

Step 2: Modify Token Serialization Functions to Support Locktime

Our functions getEncodedToken and getDecodedToken in claim-button.tsx handle converting the array of proof objects (which include properties like the secret and signatures) into a token string and back. Right now, they don't account for any custom fields like a locktime.

- **Preserve Locktime on Encoding:**
  We will update these helper functions so that when proofs are serialized, the custom locktime field is retained. This might involve, for example, ensuring that when we stringify the proofs, we do not filter out unknown keys.

- **Ensure Proper Decoding:**
  When a token is later decoded, the resulting proof objects must include the locktime property (if it was present). This change will allow us to enforce the redemption logic based on the timestamp.

Note: We need to ensure backward compatibility so that tokens minted before this change (without a locktime) are treated as immediately redeemable (i.e. default locktime set to zero).

Step 3: Enforce the Locktime Condition During Redemption

Currently , the seller manually triggers the redemption process via the "Claim" button. The redemption function (e.g. redeem()) decodes the token and then calls functions such as wallet?.checkProofsStates(proofs) to verify the proofs before redeeming funds.

Pre-Redemption Check:

We will add logic at the start of the redemption function to decode the token (using getDecodedToken()) and extract the locktime from each proof.

UI Feedback:

If the check fails, the user is alerted that the token is still locked. No further redemption steps are performed until the time lock expires.

2. Automated Processor for Payment confirmation & Shipping messages :

Below are the steps explaining my approach to implement this -

Step 1: Monitoring Order Payment and Status

Currently, When a purchase is made, state flags such as invoiceIsPaid or cashuPaymentSent are set. For example, after a successful payment, the file renders a confirmation modal:

```
{invoiceIsPaid || cashuPaymentSent ? (
  <>
    <Modal
      backdrop="blur"
      isOpen={invoiceIsPaid || cashuPaymentSent}
      onClose={() => {
        setInvoiceIsPaid(false);
        setCashuPaymentSent(false);
        router.push("/marketplace");
      }}
      // className="bg-light-fg dark:bg-dark-fg text-black dark:text-white"
      classNames={{
```

- **Monitoring Component:**
  Introduce a background monitoring routine within the listing page (or a dedicated processor module) that polls the order status at regular intervals (or subscribes to changes via a context update). This processor will detect when an order is marked as paid and then trigger the next step automatically (i.e. processing shipping confirmation).

Step 2: Automated Trigger for Shipping Message Dispatch

The current manual flow requires the seller to open a shipping modal and fill out shipping details. The onShippingSubmit function gathers data (like expected delivery date, carrier, tracking number) and then uses functions such as constructGiftWrappedEvent, constructMessageSeal, and sendGiftWrappedMessageEvent to send a shipping message:

```
const shippingCarrier = data["Shipping Carrier"];
const trackingNumber = data["Tracking Number"];
const message =
  "Your order from " +
  userNPub +
  " is expected to arrive on " +
  humanReadableDate +
  ". Your " +
  shippingCarrier +
  " tracking number is: " +
  trackingNumber;
```

- **Automated Shipping Trigger:**
  The new automated processor will monitor the order status (from Step 1) and, once the conditions are met (for example, after a preset delay or upon receiving a confirmation event from the buyer), it will automatically call the shipping message dispatch logic. This means that instead of waiting for the seller to manually click "Confirm Shipping," the processor will invoke (for instance) a function similar to onShippingSubmit but with data preloaded or triggered by business rules.

- The core functions in chat-panel.tsx that handle constructing and sending shipping messages will be reused by this processor. We may add a new helper function (or extend an existing one in nostr-helper-functions.ts) to be callable by the automated process. The processor itself could be integrated as an additional useEffect or a dedicated background hook in the order page (in [[...productId]].tsx) or even in a new file if required for clarity.

Step 3: Integration with the Messaging and Order Systems
As of now , Order page embeds the chat-based order messaging system (via <MessageFeed /> in message-feed.tsx), where shipping details are currently sent manually.

- **State Updates and UI Feedback:**
  When the automated shipping process sends out a shipping confirmation message, it will update the chats using the current mechanism already present in context.ts (e.g. via chatsContext.addNewlyCreatedMessageEvent). This ensures the buyer sees a new shipping confirmation message.

- Additionally, order status flags (e.g., a new flag like shippingConfirmed) can be updated so that the UI in both the listing and order pages reflects that shipping information has been dispatched automatically.

3.UI To handle Disputes between Seller & Buyer:
Below are the steps explaining my approach to implement this -

Step 1: Identify the UI Anchor in the Order/Chat Flow
The order and messaging interface is handled mainly in orders/index.tsx (which renders the <MessageFeed />) and messages/chat-panel.tsx. There is no dedicated dispute feature at present.

- Introduce a new "Raise Dispute" trigger/button into the chat panel (or in the order detail view) so that either party (buyer or seller) can initiate a dispute.
  For example, within chat-panel.tsx, below the shipping confirmation button, add a "Raise Dispute" button.

```
            <Button className={SHOPSTRBUTTONCLASSNAMES} type="submit">
              Confirm Shipping
            </Button>
          </ModalFooter>
        </form>
      </ModalContent>
```

Step 2: Create a Dispute Initiation Modal

Right now , we already have UI modals in our code (shipping modal and review modal in chat-panel.tsx).

- Develop a new modal , call it the "Dispute Modal"—that pops up when the "Raise Dispute" button is clicked.
- This modal should include simple input fields:
- A dropdown or a set of buttons to choose the dispute reason (e.g., "Non-delivery", "Product quality issues", etc.).
- A text area for additional comments and File upload for optional evidence.
- Use components such as the <Modal>, <ModalContent>, <ModalBody>, and <ModalFooter> (already used in chat-panel.tsx) to build the modal.
- The outcome of this modal will be a structured dispute object that can be attached as a new message in the chat stream.

Step 3: Integrate the Dispute Message Flow into Chat

The chat messages are handled in chat-panel.tsx. New messages are added via the context function ChatsContextInterface (addNewlyCreatedMessageEvent….).

- After the dispute form in the modal is submitted, use a helper function (new one or maybe existing functions such as constructGiftWrappedEvent can be repurposed) to build a dispute message event.
- Call chatsContextInterface (and thus addnewlyCreatedMessageEvent) to update the chat with a new message indicating a dispute has been raised. (E.g., a message with a type or tag like "dispute" can be used to distinguish it from other chat messages.)

- Ensure that the dispute message has clear labels (e.g., "Dispute Raised by Buyer" or "Dispute Raised by Seller") so both parties understand the current state.

Step 4: Apply the 2-of-3 Multisig Concept for Dispute Escrow (Moderation Logic)

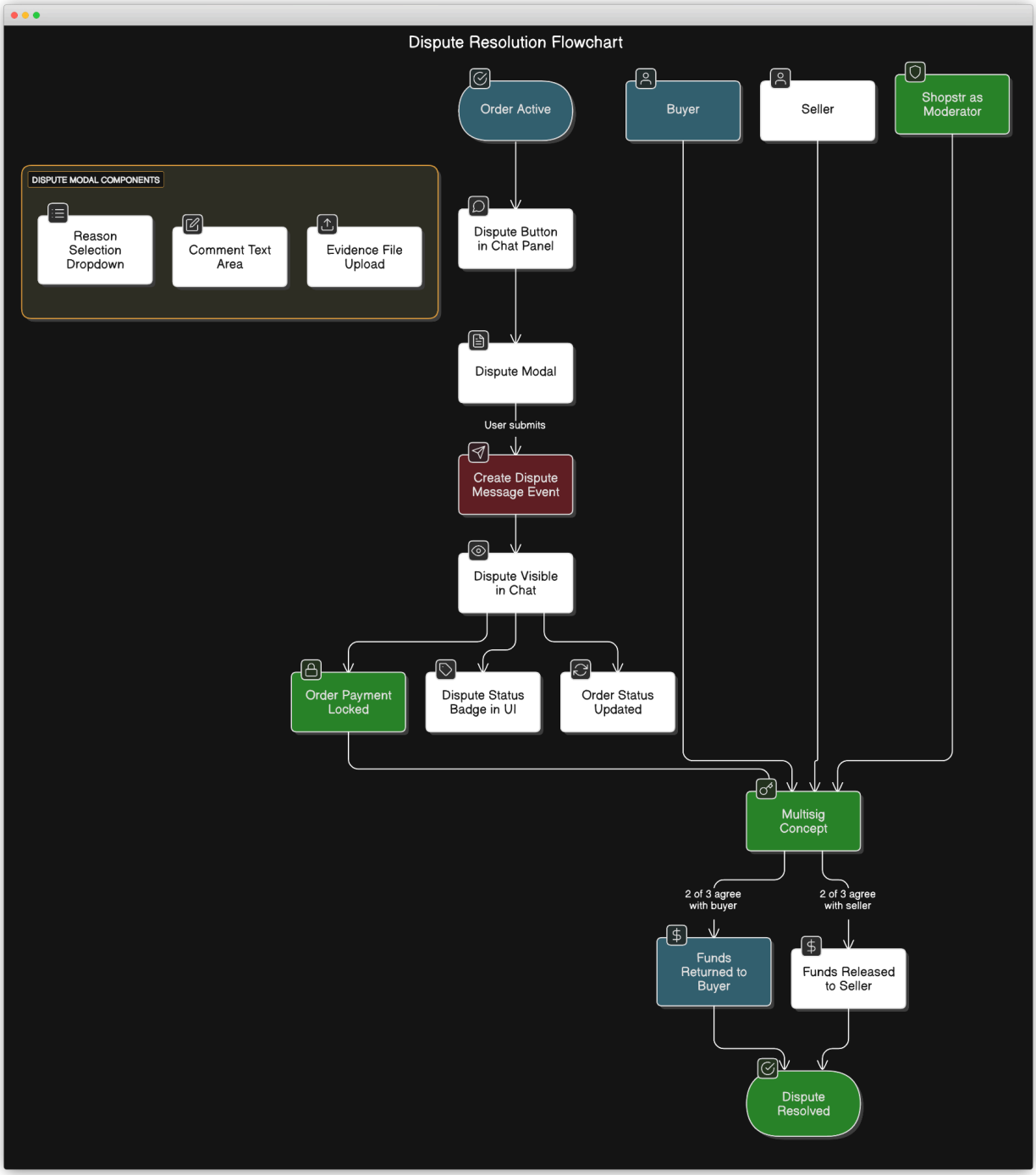
- **2-of-3 Multisig Principle:** In a typical multisig escrow, funds are held in an address that requires approval (via signatures) from two out of three parties: the buyer, the seller, and an independent moderator.
- Although Shopstr does not currently deploy a full multisig backend, our planned enhancement is to adopt the 2-of-3 concept in the dispute resolution process. For now, Shopstr will serve as the de facto moderator.
- So , we will add logic - when a dispute is raised, create a dispute event that conceptually "locks" the payment. Shopstr automatically participates as the third party in the dispute escrow. The event data (sent as part of the dispute message) will include extra fields or tags indicating the expected roles (e.g., buyer, seller, moderator) and their respective signatures/approvals.
- Helper functions from nostr-helper-functions.ts may be extended or used to incorporate the additional tags required by a 2-of-3 multisig approach.

Note: Although the actual cryptographic enforcement (for example, using a multisig scheme to ensure funds are released only when two of three valid signatures are present) may be added later, for now our UI will clearly indicate the dispute status and rely on Shopstr's moderation decision communicated via subsequent dispute resolution messages.

UI Feedback and State Management for Disputes

- Update the UI to display dispute status prominently. This could be a badge or a highlighted message in the chat when a dispute is active.
- Optionally, add an indicator (like a status label) in the order summary (in listing/[[...productId]].tsx or orders/index.tsx) showing the dispute state.
- For now, since Shopstr acts as the sole moderator, ensure that once a dispute is raised, the UI guides the user that further resolution will be managed by Shopstr. This clarification should be part of the dispute modal as well as the resulting dispute message.

Below Flowchart helps to understand workflow we might have better-



4. Add HODL Invoices as another option to hold funds (Stretch goal)

Below steps might be taken to implement this if all other features are implemented and time permits (Otherwise , I think we can slowly work on implementing this after SOB end because i am not aware of its complexity as of now).

Step 1: Creating a Hold Invoice

Shopstr's current payment flow uses Cashu tokens with a manual claim mechanism. For HODL invoices, we will generate a 32-byte preimage (using SHA-256 on arbitrary input) and derive its hash. Then we call LND's AddHoldInvoice API with parameters such as memo, value, expiry, and the preimage's hash.

- **Preimage Generation:** Generate a preimage and its hash.
- **API Call:** Use the LND AddHoldInvoice API (as per the official documentation) to create the hold invoice.

Step 2: Secure Storage & Server-Side Monitoring

To ensure the preimage isn't exposed to unauthorized parties:

- **Encryption:** The generated preimage will be stored encrypted in a secure backend store instead of in plaintext on the client-side.
- **Access Control:** Only the automated settlement module (which may run on a secure server) can decrypt and use it when the conditions are met.

Server-Side Implementation:

Rather than handling invoice monitoring solely on the client, we will move the critical parts of monitoring and settlement logic to a secure server process or background service. This approach:

- Increases reliability by ensuring that even if a user closes their browser, the invoice state is still being tracked.
- Provides better security controls around the preimage and API calls.

Step 3: Monitoring Invoice State

Currently, the order page and messaging system handle manual triggers via UI modals for shipping and payment confirmation.

- **Subscribe to Invoice Updates:**The server-side process will subscribe to hold-invoice state updates using LND's SubscribeSingleInvoice API.

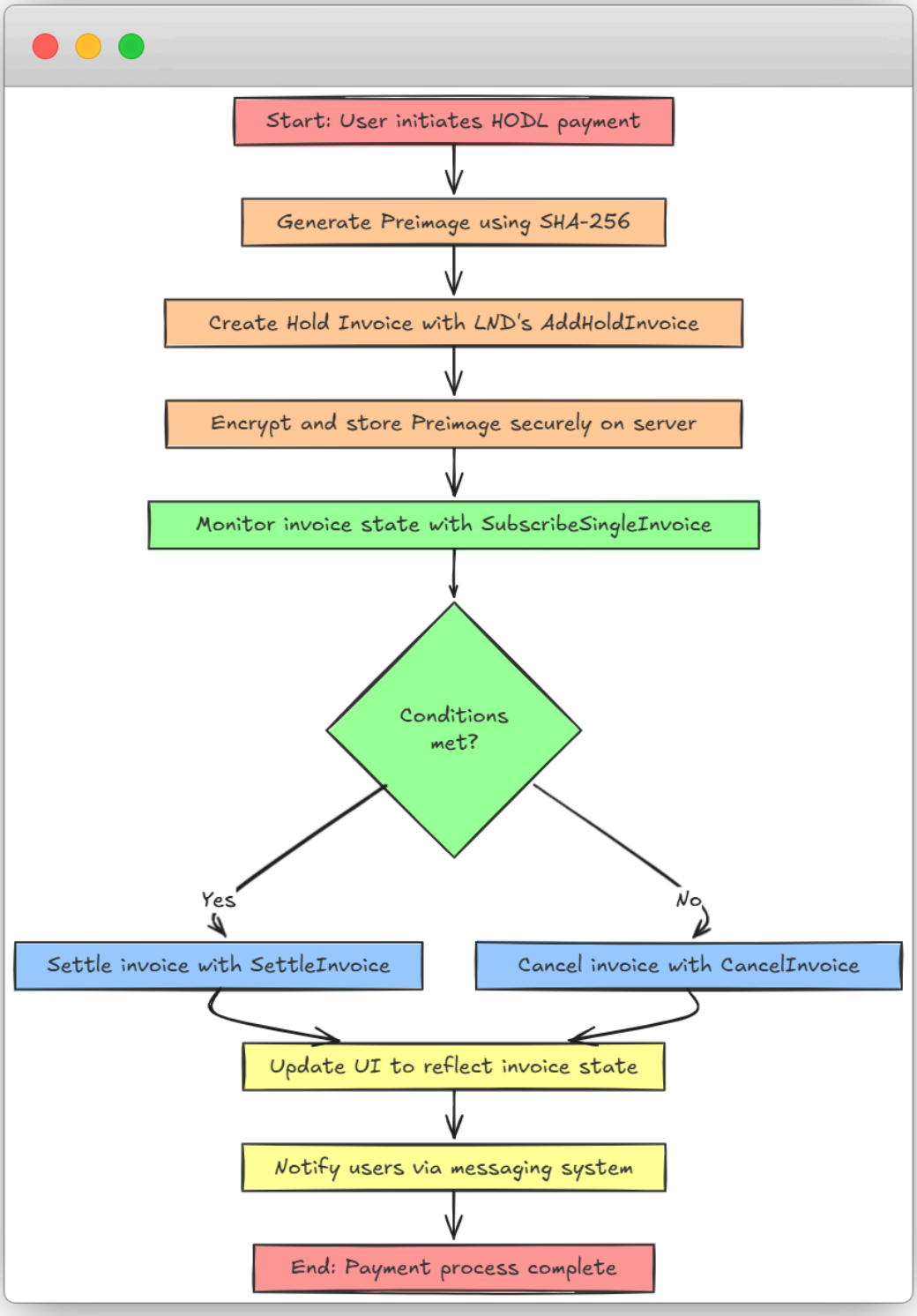
- **Periodic Checks:** Alternatively, it may poll or listen to events so that when the invoice's state changes (e.g. remains held until conditions are met or if it is manually cancelled), the system can update the order state in our database and notify the relevant parties.
- **State Updates:** This process will update the order history and notify the buyer/seller (for example, through existing messaging APIs) once the invoice is settled or cancelled.

Step 4: Automated Settlement of the Invoice

- **Criteria Definition:** Define the conditions for settlement—such as after confirming order fulfillment or after a designated timeout.
- **Preimage Release:** When conditions are met, the secure server process uses the decrypted preimage (protected as described in Step 2) to call LND's SettleInvoice API. If conditions aren't met after a timeout, the process may instead call LND's CancelInvoice API.
- **UI Update:** Once the invoice is settled, the server sends an update (for example, via our existing messaging or context update methods) to refresh the order status.

Step 5: Updating the User Interface

- **Invoice Status:** Modify UI components such as the order summary (in [[...productId]].tsx) or transaction history (transactions.tsx) to display a "Payment Held" status.
- **Notifications:** Once the server-side process automatically settles the invoice, the UI is updated through our messaging system (using context functions like addNewlyCreatedMessageEvent, which is defined in context/context.ts) so that both buyer and seller see the updated status (for example, "Funds Released" or "Shipping Confirmed").


Start: User initiates HODL payment
Generate Preimage using SHA-256
Create Hold Invoice with LND's AddHoldInvoice
Encrypt and store Preimage securely on server
Monitor invoice state with SubscribeSingleInvoice
Conditions met?
Yes
No
Settle invoice with SettleInvoice
Cancel invoice with CancelInvoice
Update UI to reflect invoice state
Notify users via messaging system
End: Payment process complete

Timeline and Deliverables:

Before Program Starts [25 April – 15 May]

- Understand Shopstr codebase in detail and go through Cashu documentation.
- Review some pr's if raised during this period and try to understand them.
- Familiarize myself with the Nostr event system used for messaging in Shopstr

P2PK Time-Locked Cashu Tokens [15 May – 29 May]

- Review the minting flow to understand the current process.
- Implement P2PK locking mechanism to secure tokens to recipient's public key.
- Extend proof objects by adding a new locktime property.

P2PK Time-Locked Cashu Tokens [29 May – 12 June]

- Update token serialization functions (getEncodedToken/getDecodedToken) to retain the locktime.
- Integrate the redemption process to check against the locktime.
- Develop unit tests to ensure proper encoding and timely redemption.

Payment Confirmation & Shipping [12 June – 26 June]

- Design a background monitoring routine to track order status changes.
- Implement initial logic to automatically trigger the shipping confirmation once payment is confirmed.
- Document the processor's integration within the existing order page.

Payment Confirmation & Shipping [26 June – 10 July]

- Refine and integrate the auto-triggered shipping message using existing messaging functions.
- Update UI state via chatsContext to reflect shipping status.
- Conduct integration testing to ensure smooth order processing.
- Plus, mid-term evaluation takes place.

Dispute Resolution UI [10 July – 24 July]

- Implement dispute functionality with a "Raise Dispute" button and modal containing reason selection, comments, and evidence upload options.
- Integrate structured dispute objects into the chat stream with clear UI indicators showing dispute status.
- Design and implement the dispute resolution workflow using a 2-of-3 multisig approach with Shopstr as the moderator.

HODL Invoices & Project Finalization [25 July – 15 August]

- Implement HODL invoices functionality with preimage generation and secure storage mechanisms.
- Integrate HODL invoice monitoring with the existing order processing and payment systems
- Utilize final week for refinement ,bug fixes, and documentation completion(kind of report of work I did) as a buffer period.

Post SOB:

While I aim to complete everything within the program's timeframe, if any unforeseen challenges arise, I'll be more than happy to finish the remaining work after the conclusion of SOB. Additionally, I'd love to create a developer guide in my free time as an intern to help future contributors better understand the codebase and the Bitcoin e-commerce system.

Some Contributions to Shopstr:

#PR1 , #PR2 , #PR3 , #PR4 , #PR5 , Bug Reported

Competency test:

Repo , Website