WHITEPAPER · VERSION 0.6 · 26 SEPTEMBER 2026

Levy: pull-based subscription billing in USDC on Arc

How Levy collects recurring payments from a one-time USDC allowance, what the contract guarantees to customers and merchants, and what it does not.

Status. Everything described here is built and tested, and the contracts have been live on Arc testnet since 26 September 2026 for a week-long soak (Appendix D). Nothing is on mainnet and nothing is audited yet. The paper describes the contracts as of commit `08dd7b7`, which is the code deployed, and the services as of the same commit. Figures from our own measurements say how they were taken.

ABSTRACT

Software is sold by subscription, yet stablecoin payments still work like invoices: the customer has to act every time a payment is due. Levy is a protocol for subscription billing in USDC on Arc. A customer approves a USDC allowance once, sized in billing cycles they choose. From then on a smart contract lets anyone trigger each payment when it falls due, and it can only collect the plan's price, once per billing date, for the plan's merchant. A failed payment is recorded on-chain instead of reverting. That lets an off-chain keeper run retries while the contract itself decides when a subscription is past due and when it pauses. The merchant receives 99% of each payment in the transaction that collects it; 1% goes to the protocol. Around the contract, Levy provides the services merchants expect from card billing: signed webhooks, invoice records, a dashboard, a customer portal and a hosted checkout. This paper describes the design, its accounting and security properties, its economics and its known limitations.

CONTENTS

1. Introduction

1.1 Subscriptions and the rails under them

Most subscriptions run on cards. A card on file lets a merchant charge the customer every cycle without asking again, and that convenience is expensive. Stripe's published US pricing is 2.9% plus 30 cents per domestic card payment, 1.5% more for international cards and $15 per dispute, and its Billing product adds 0.7% of billing volume [1]. On a $29 monthly plan those list prices come to about $1.34 per payment, or 4.6%, before any dispute. A card payment can also be reversed weeks later through a chargeback.

Stablecoins remove most of that cost. A USDC transfer settles in seconds, is final and costs a fraction of a cent on a chain built for it. But a token transfer is a push payment: the customer's wallet has to sign each one. For a subscription, that means asking the customer to act every cycle, and every missed signature is a failed renewal. Merchants who accept stablecoins today mostly send an invoice each cycle and wait.

1.2 Earlier approaches

Recurring payments on EVM chains have been proposed before. The EIP-948 discussion and ERC-1337 (2018) described subscriptions in which the customer signs a replayable authorisation that the merchant later submits to the customer's smart-contract wallet [7, 8]. ERC-1337 is marked stagnant, and its design needs a contract wallet, which most customers do not have. Streaming protocols take another route and move value continuously, second by second. That suits payroll and vesting. Software billing, though, is priced per cycle, invoiced per payment and reconciled per customer, and merchants expect retries, prorated plan changes and a record of every charge.

1.3 Levy's approach

Levy builds on a primitive every ERC-20 token already has: the allowance [5]. The customer approves the Levy contract to spend an amount of USDC they choose (the checkout sizes it as a number of billing cycles) and subscribes to a plan. From then on the contract decides what may be pulled and when: at most the plan's price, once per billing date, for the plan's merchant. Anyone may trigger a payment that is due, and Levy runs a keeper that does so on schedule. A failed payment is written to the chain instead of reverting, so the contract, not an off-chain service, decides when a subscription is past due and when it stops. Around this contract Levy provides what merchants expect from card billing: dunning, signed webhooks, invoice records, a dashboard, a customer portal and a hosted checkout.

1.4 Why Arc

Arc is a layer-1 chain built around USDC. Four of its properties matter for billing:

- **USDC is the gas token.** Customers, merchants and the keeper pay network fees in the same dollars they bill in. Nobody has to hold a second token.
- **Fees are low and stable by design.** Arc prices gas with EIP-1559 plus a moving average of block utilisation, targets about $0.001 for an ERC-20 transfer and caps the base fee [2, 12]. A billing operator can predict its costs.

- **Finality is deterministic and takes under a second.** Blocks are finalised by Malachite BFT consensus and are never reorganised [3]. A charge is final once it is mined, so the keeper needs no confirmation depth and a merchant can act on the webhook at once.
- **It is EVM-compatible.** USDC has an ERC-20 interface with 6 decimals at `0x3600…0000`, and Permit2 is deployed at its canonical address [4, 6].

2. Design goals

Levy was designed against five goals. Later sections show how each is met.

1. **Bounded exposure.** A customer can work out the most the contract will ever take from them: never more than their allowance, and never more than one plan price per billing date (§6.2).
2. **No custody.** Levy never holds funds at rest. Each payment is split between the merchant and the fee router in the transaction that collects it.
3. **Permissionless execution.** The payment trigger is open to anyone. Levy's keeper is needed for timeliness, not for correctness: if it stops, charges wait, and nothing is lost or double-charged.
4. **Merchant-grade operations.** Trials, dunning, prorated plan changes, invoice records, signed webhooks and customer references are part of the system, not left to each merchant.
5. **Exact accounting.** Integer USDC units, rounding in a documented direction, and invariants checked by fuzzing.

Out of scope: converting USDC to fiat, tax calculation, on-chain refunds, usage-based billing and currencies other than USDC.

3. System overview

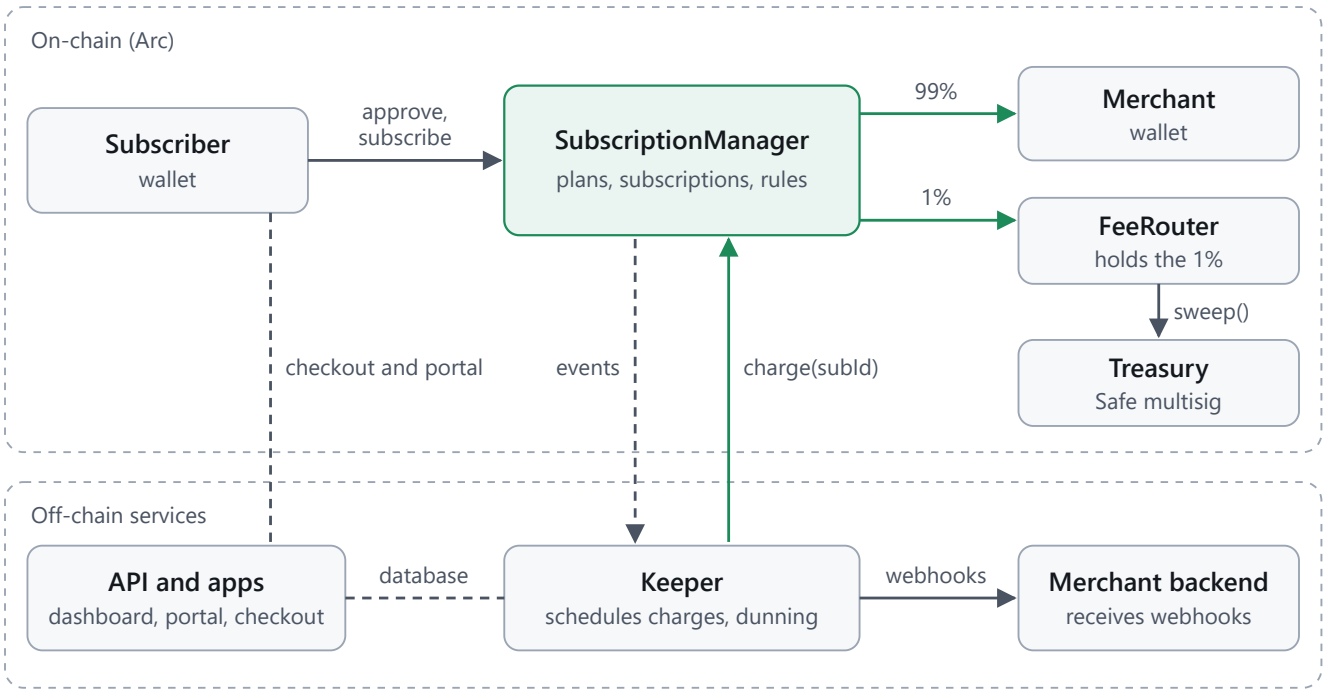

Figure 1. Payments move only on-chain (green arrows), from the subscriber's allowance through the SubscriptionManager to the merchant and the FeeRouter in one transaction. Off-chain services decide when to call `charge` and tell merchants what happened.

There are four actors:

- **Merchant:** creates plans from its wallet and receives 99% of each payment.
- **Subscriber:** approves a USDC allowance, subscribes, and can pause, resume or cancel.
- **Keeper:** an off-chain service that calls `charge` when payments fall due. Levy runs one; anyone can.
- **Treasury:** receives the protocol fee from the fee router. On mainnet it is a Safe multisig.

The split between on-chain and off-chain follows one rule: anything that decides who pays what, and when, lives in the contract. Anything that only affects timing or communication lives off-chain.

Concern	Where it lives
Plan terms and prices	Contract
Who may be charged, how much, from which date, and to whom	Contract
Failure count, past-due and auto-pause thresholds	Contract
The 99/1 split	Contract
When to call `charge`; the retry schedule	Keeper

Concern	Where it lives
Invoice records, metrics, allowance warnings	Keeper database and API
Notifying merchants	Signed webhooks
Emailing customers, refunds, tax	The merchant

4. Protocol

The protocol is two contracts: `SubscriptionManager`, which holds plans and subscriptions and moves payments, and `FeeRouter`, which receives the protocol fee. Appendix A lists their interfaces.

4.1 Units

All amounts are 6-decimal USDC units through the ERC-20 interface: 1 USDC is 1,000,000 units. Arc's native gas balance uses 18 decimals, and those values never enter payment logic. Times are chain timestamps in seconds.

4.2 Plans

A merchant creates a plan with `createPlan(price, cycleSeconds, trialSeconds, graceSeconds, maxFailed)` and becomes that plan's merchant.

Parameter	Meaning	Rule
`price`	Price per cycle, in units	Above 0
`cycleSeconds`	Length of a billing cycle	1 day to 10 years
`trialSeconds`	Free trial before the first charge	Shorter than the cycle; 0 for none
`graceSeconds`	Informational	Not enforced; retries are scheduled off-chain
`maxFailed`	Failed charges before the subscription is past due	1 to 255

Plans are immutable; there is no function to edit one. A merchant who wants a new price creates a new plan, and existing subscribers keep the old price unless they choose to switch (§4.8). The one-day minimum cycle allows each subscription at most one billing date per day, which limits how quickly a mistaken approval could be spent (§6.5). Cycles are fixed lengths in seconds, so a 30-day "month" drifts against calendar months.

4.3 Subscriptions

A subscription records its `subscriber`, `planId`, `start`, `nextChargeAt` (the next billing date), `failedAttempts`, proration `credit`, `chargeCount` and `status`. The status is one of TRIAL, ACTIVE,

PAST_DUE, PAUSED or CANCELLED (Figure 2). A separate mapping, `subscriptionRef`, holds the merchant's 32-byte reference for the customer (§4.10).

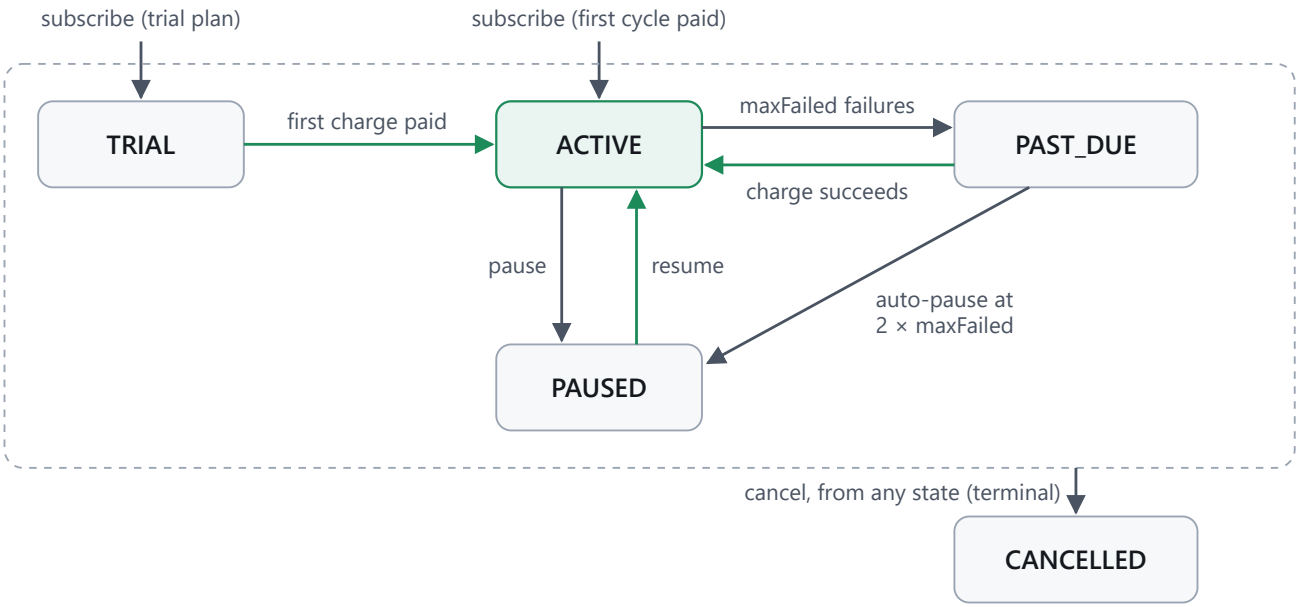

Figure 2. Subscription states. Pause is also available from TRIAL and PAST_DUE, and a trial whose first charge keeps failing reaches PAST_DUE the same way an active subscription does. Only CANCELLED is final.

4.4 Subscribing

The subscriber calls `subscribe(planId, ref)`. The subscriber is always the caller, so nobody can subscribe someone else. If the plan has a trial, the subscription starts in TRIAL and its first billing date is the end of the trial. Otherwise billing is in advance: the first cycle is collected in the same transaction, the next billing date is one cycle later, and if the payment fails the whole subscribe reverts. A customer either pays for the cycle they are entering or does not subscribe.

There are two ways to authorise payments:

- **Approve, then subscribe.** An ERC-20 `approve` to the SubscriptionManager, then `subscribe`: two transactions. The hosted checkout walks the customer through both and suggests 12 cycles' worth, added to any allowance they already gave Levy. Levy's own interfaces never ask for an unlimited allowance.
- **Permit2.** `subscribeWithPermit2(planId, ref, permit, signature)` grants the allowance through Permit2 [6] with an EIP-712 signature [9] and subscribes in one transaction. The contract checks that the permit names itself as spender and USDC as the token. Only the signer can submit it: the permit does not name a plan, so a third party who could relay it could also choose the plan.

4.5 Charging

Once a subscription's billing date has passed, anyone may call `charge(subId)`:

```
charge(s):
  require s.status in {TRIAL, ACTIVE, PAST_DUE}
  require now >= s.nextChargeAt
  if collect(s, plan):
      s.nextChargeAt += plan.cycleSeconds      // from the old date, not from now
      s.failedAttempts = 0
      if s.status in {TRIAL, PAST_DUE}: s.status = ACTIVE
  else:
      require now >= lastFailedAt[s] + 1 day  // else revert RetryTooSoon
      lastFailedAt[s] = now
      s.failedAttempts += 1                    // emits ChargeFailed; no revert
      if s.failedAttempts >= maxFailed:     s.status = PAST_DUE
      if s.failedAttempts >= 2 * maxFailed: s.status = PAUSED   // emits AutoPaused

collect(s, plan):                               // shared by subscribe, charge, changePlan
  if s.credit >= plan.price:
      s.credit -= plan.price; due = 0; fee = 0 // cycle paid from credit
  else:
      due = plan.price - s.credit
      if not pull(s.subscriber, due): return false   // nothing changed
      s.credit = 0
      fee = floor(due / 100)
      transfer(merchant, due - fee)
      transfer(feeRouter, fee)
  s.chargeCount += 1                            // emits Charged
  return true
```

Some consequences of this design:

- **No funds at rest.** The pull goes into the contract and straight back out to the merchant and the fee router in the same transaction.
- **The schedule does not drift.** A successful charge moves the billing date forward by exactly one cycle from its previous value. A late call does not shift later dates. If the keeper misses several billing dates, each needs its own call and each collects one cycle.
- **Failures are recorded, not reverted, and at most once a day.** A pull fails when the subscriber lacks the balance or the allowance. The call still succeeds, the failure count goes up and `ChargeFailed` is emitted, so the failure is on the record and the keeper's transaction is not wasted on a revert. Because anyone can call `charge`, the contract records at most one failure per subscription per day: a failing call sooner reverts with `RetryTooSoon` and changes nothing, so nobody can rush a subscription through dunning. Successful charges are never held back.
- **Two allowance sources.** The pull tries the direct ERC-20 allowance first, then a Permit2 allowance, and stops at the first that succeeds.
- **Every charge has an invoice reference.** `Charged(subId, amount, fee, invoiceRef, chargeCount)` carries `invoiceRef = keccak256(abi.encode(subId, chargeCount))`, a deterministic reference unique to that charge.

4.6 Dunning

Dunning, the process of recovering failed payments, is split between the chain and the keeper. The contract holds the thresholds: at `maxFailed` failures a subscription becomes PAST_DUE and can still be charged; at twice that, `charge` itself pauses it. No off-chain service decides to stop billing a customer, and because failures are recorded at most a day apart, nobody can make that happen faster than the thresholds allow: a plan with `maxFailed` of 2 cannot pause itself sooner than three days after the first failure. The keeper holds the schedule: by default it retries 1 day after the first failure, 3 days after the second and 7 days after each later one (configurable per deployment). A success at any point resets the failure count, returns the subscription to ACTIVE and moves the billing date one cycle past the missed one.

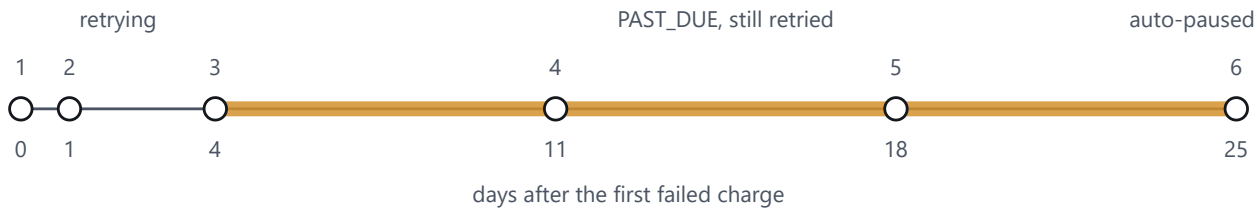

Figure 3. A plan with `maxFailed` = 3 under the default retry schedule, assuming no retry succeeds. Numbered dots are failed attempts. The subscription is past due 4 days after the first failure and pauses itself after 25.

Merchants receive `charge.failed`, `dunning.started` and `subscription.paused` webhooks and contact the customer themselves. Levy sends no email to subscribers.

4.7 Pause, resume and cancel

Both the subscriber and the merchant can pause, resume and cancel a subscription. Nothing is charged while it is paused, and days already paid for keep running. Resuming is a fresh start: if the billing date passed during the pause, it moves to the moment of resuming, so paused cycles are never billed and an overdue cycle is forgiven, and the failure count and the one-day failure timer reset. Cancelling is final: a cancelled subscription can never be charged or resumed.

> Because the merchant can also resume, a pause is shared between the two parties, not a lock the customer holds. A customer's hard stops are cancelling and reducing or revoking the allowance. Refunds are the merchant's policy, paid from the merchant's wallet; the contract has no refund function.

4.8 Plan changes and proration

A subscriber can move to another plan from the same merchant with `changePlan(subId, newPlanId)`. Only the subscriber can call it, and a plan from a different merchant is refused. The new plan starts and is billed at once, less credit for unused paid time on the old plan:

```
r      = min(nextChargeAt - now, cycle_old)   if chargeCount > 0 and nextChargeAt > now
       = 0                                    otherwise
credit = credit + floor(price_old * r / cycle_old)
```

The contract then collects the new plan's first cycle, spending credit before pulling anything, sets the next billing date one new cycle ahead, resets the failure count and marks the subscription ACTIVE. If the pull fails, the whole change reverts.

These rules keep credit honest. It is earned only for time that was paid for, so trial time earns none. One change adds at most one old-plan price. It is added to existing credit rather than replacing it. It is only ever spent with the merchant who received the USDC behind it. The invariant suite checks that a subscription's credit never exceeds what it has paid (§5.2).

- **Upgrade.** Ten days into a $30, 30-day plan, the customer moves to a $60, 30-day plan. The 20 unused days become $20 of credit, the contract pulls $40, and the next billing date is 30 days away.
- **Downgrade.** Ten days into a $60, 30-day plan, the customer moves to a $10, 30-day plan. The 20 unused days become $40 of credit. The first $10 cycle is paid from it, and the remaining $30 covers the next three cycles, each recorded as a $0 charge.
- **Overdue.** A change made after the billing date has passed earns no credit and forgives the overdue cycle, as cancelling and resubscribing would.

4.9 Fees

The protocol fee is `floor(due / 100)`: 1% rounded down, with any fraction of a unit left to the merchant. A pull under 100 units ($0.0001) pays no fee, and a cycle paid entirely from credit pays none. The rate is a constant in the contract, not a plan parameter, so changing it takes a new deployment. Merchants can rely on it. Fees reach the `FeeRouter` in the charging transaction, and its permissionless `sweep()` forwards the balance to the treasury. Only the current treasury can name a new one.

4.10 Merchant references

A merchant's checkout link can carry a reference for its own customer record: 32 bytes, entered in the checkout as up to 31 ASCII characters or 64 hex digits. It is stored with the subscription and included in every webhook. The reference is public, and the subscriber supplies it when subscribing, so it is a claim rather than proof. Someone could copy a victim's reference into a subscription of their own and cancel it. Merchants should issue unguessable references per checkout and, once a subscription is created, track it by its `subId`.

5. Accounting

5.1 Rounding

The contracts use integer arithmetic only. They never convert decimals and never touch native 18-decimal values. The deploy script refuses a USDC address that does not report 6 decimals. Two operations divide, and both round in the merchant's favour by less than one unit:

Operation	Formula	Remainder (under 1 unit) goes to
Protocol fee	`floor(due / 100)`	Merchant
Proration credit	`floor(price * r / cycle)`	Merchant, who already collected the full price

Timestamps are 64-bit, far beyond any realistic date. A multiplication in the proration formula could only overflow for prices above about 3.7×10^{68} units, and Solidity 0.8 reverts on overflow rather than wrapping.

5.2 Invariants

Foundry's invariant fuzzer runs random sequences of plan creation, subscribing, approvals, charges across warped time, plan changes, pauses, resumes, cancellations and fee sweeps, with and without trials and across the full `maxFailed` range of 1 to 255, plus a griefer that calls `charge` several times at the same moment. After every step it checks eight properties:

1. **Conservation:** every USDC unit minted in the test sits in exactly one tracked account.
2. **No funds at rest:** the SubscriptionManager's balance is always zero.
3. **Bounded failures:** `failedAttempts` never exceeds twice `maxFailed`.
4. **Spaced failures:** a subscription's recorded failures are never less than a day apart.
5. **Cancel is terminal:** a cancelled subscription's charge count never changes again.
6. **Monotonic schedule:** a billing date never falls before the subscription's start.
7. **Credit is backed:** a subscription never holds more credit than the USDC it has paid.
8. **Credit is bounded:** credit never exceeds the largest plan price.

Continuous integration runs 1,000 sequences of 40 calls on every push, alongside 53 unit tests. The keeper has 55 tests, including an end-to-end run against a local chain; the API has 32.

6. Security model

6.1 Trust assumptions

Party	Can	Cannot
Merchant	Create plans; pause, resume or cancel its own subscriptions	Change a price, move a customer to another plan, charge a wallet that did not subscribe, charge before a billing date, or undo a cancellation
Levy's keeper	Choose when a due charge is submitted	Change amounts, recipients or dates; charge early; receive anything from a charge

Party	Can	Cannot
Anyone else	Call `charge` on a due subscription; call `sweep`	Redirect or receive funds
Treasury	Receive fees; name a new treasury	Touch payments or subscriptions
Levy's API and webhooks	Show data; send notifications	Move funds
USDC issuer	Freeze addresses. A frozen merchant or fee router makes charges on the affected plans revert, and the customer is not charged	Nothing specific to Levy
Arc validators	Order and finalise transactions; a halt delays charges	Change the contract's rules

6.2 Customer exposure

The contract pulls from a customer in exactly three places. `subscribe` and `changePlan` can only be called by the customer. `charge` can be called by anyone, but each successful call needs a billing date that has passed, collects at most one plan price and moves the billing date exactly one cycle later. The only other writes to the billing date are resume, which can only move it later, and the customer's own plan change. It follows that for a subscription with price p, cycle c and first billing date d_1, the total that other parties can cause to be pulled by time t is

```
pulled_by_others(t) <= p * (1 + floor((t - d1) / c))      for t >= d1, and 0 before d1
```

A plan change by the customer starts a new schedule under the same rule. Across all of a customer's Levy subscriptions, whichever merchants they belong to, the total can never exceed the allowances the customer granted to the SubscriptionManager, directly or through Permit2, because every pull spends them. This bound follows from the code. It is not one of the fuzzed invariants, and we would like the audit to check it.

6.3 Why the charge function can be public

Consider `charge` called by an attacker. The subscription and plan are read from storage, so the caller influences nothing. Tokens move from the subscriber to the contract, then to the plan's merchant and the fee router, both fixed. The caller receives nothing, and calling first, or front-running the keeper, changes nothing but who paid the gas. A caller can charge a due subscription at the earliest moment it is due, which is the intended behaviour, and can record a failure when a pull fails, but no more than one a day per subscription (§4.5). Before that limit existed, a caller could record failures back to back and pause a subscription in one block; see §6.8.

6.4 Reentrancy

Every function that moves tokens is guarded against reentrancy. State is written after the transfers in these functions, the reverse of the usual checks-effects-interactions order. That is safe with USDC, which makes no callbacks on transfer, but it would matter with a token that did, and pause, resume and cancel are not guarded. This is flagged for the audit.

6.5 Denial of service and keeper cost

A customer who revokes their allowance cannot make the keeper's transaction revert; the failure is recorded and costs one transaction. Each subscription can fail at most twice `maxFailed` times before it pauses itself, at most once a day, and only a resume, which the attacker must pay for, buys another round.

A cheaper attack is a self-funded subscription: create a plan priced at one unit and subscribe to it. The attacker pays themselves, so charges never fail and never pause, and the keeper pays gas for each one. The one-day minimum cycle limits every such subscription to one charge a day, but the number of subscriptions is unbounded. The mitigation is keeper policy, for example skipping subscriptions whose fee does not cover their gas (§8.4), and the keeper's gas spend is on the monitoring list.

6.6 Permit2

An earlier version let anyone relay a customer's Permit2 signature. Because a permit does not name a plan, a relayer, or anyone who read it from the mempool, could have subscribed the customer to a plan of their own. Only the signer can submit it now. A remaining low-severity issue: someone who sees the signed permit can submit it to Permit2 directly and use up its nonce, making the customer's subscribe fail. No funds are at risk, the allowance is still granted, and the customer can subscribe the ordinary way.

6.7 Off-chain services

The API authenticates with Sign-In with Ethereum [10]. Messages are bound to the deployment's origin, challenges are single-use, and sessions last 24 hours and are stored hashed. The dashboard and portal hold the session in a cookie that page scripts cannot read, and a write carried by that cookie must come from the app's own origin. Every endpoint is scoped to the signed-in address, and another merchant's records return not-found. Webhook URLs must use https on port 443 at a public address, checked at registration and again at connection time so a DNS change cannot redirect deliveries to an internal network, and redirects are never followed. Deliveries are signed with Ed25519 [11] over the timestamp, delivery id, event type and body, so a captured delivery cannot be replayed later or relabelled as a different event. Delivery runs in its own loop, so a slow merchant endpoint cannot delay charging. The dashboard, portal and checkout send a Content-Security-Policy that allows scripts only from the app's own origin, so an injected script cannot run.

6.8 Findings so far

Code review, fuzzing and an external engineering review found six issues. All are fixed and have regression tests:

1. **Unbacked proration credit.** Billing used to be in arrears, and a plan change credited unused time on a cycle nobody had paid for, from any merchant's plan. A customer could mint credit on a plan of their own and spend it on a real merchant's plan. Fixed by billing in advance, earning credit only from paid time and allowing same-merchant changes only.
2. **Relayed Permit2 subscriptions** (§6.6).
3. **Resume billed paused cycles** and kept the failure count. Resume is now a fresh start.
4. **An overflow in the auto-pause check.** For `maxFailed` of 128 or more, doubling it in 8-bit arithmetic overflowed and made every failed charge revert. The fuzzer's range was widened to 1 to 255.
5. **No minimum cycle.** A one-second plan let a customer's mistaken approval be drained in seconds and let spam subscriptions cost the keeper gas every 30 seconds. The minimum is now one day.
6. **Failures recorded back to back.** Found by an external review after the first testnet deployment. Since `charge` is permissionless and a failed pull does not move the billing date, anyone could call it repeatedly while a customer's payment was failing and pause the subscription in one block, skipping the dunning schedule. No funds could move. Failures are now recorded at most once a day (§4.5), the review's attack is a regression test, and the fuzzer checks it with a griefing action. The contracts were redeployed on testnet before the soak.

Open items, all tracked for the audit and the testnet soak:

- Self-funded spam subscriptions (§6.5).
- The inverted update order (§6.4).
- Plan changes and resumes forgive an overdue cycle. This is intended, but merchants should know it.
- The Permit2 nonce front-run (§6.6).
- A frozen merchant blocks charges on its plans.
- The treasury can hand itself to a single-key address.
- The off-chain gaps listed in §9.

7. Off-chain services

7.1 Keeper

The keeper runs two independent loops. The charge loop, every 30 seconds:

1. Syncs contract events into its database.
2. Charges due subscriptions.
3. Runs dunning retries.
4. Backs off charges that revert for reasons other than payment (after 1, 5 and 15 minutes, then 1 hour, then every 6 hours).
5. Checks allowance runway.

The delivery loop, every 5 seconds, sends webhooks. All schedule decisions use the chain's timestamp, not the server clock. Because a block is final, the keeper treats a mined transaction as settled.

Indexing is deliberately narrow. On Arc, USDC is the gas token, so USDC logs are everywhere. We measured about 4,000 of them per 500 blocks on testnet and about 1,200 on mainnet on 25 September 2026, roughly 1.4 million and 430,000 a day. The keeper fetches only Levy's own events, USDC transfers into or out of the SubscriptionManager and approvals to it, in fixed-size block ranges. It sends those queries one at a time and backs off for up to 15 seconds when refused, because Arc's public testnet RPC rate-limits bursts of them.

The allowance check reads up to 50 subscribers per cycle, rechecking each after 6 hours, or sooner after a charge or a new approval. When a subscriber's allowance covers fewer than two more rounds of all their subscriptions, each affected merchant gets an `allowance.low` webhook. It is sent once, and again only if the allowance recovers and runs low again.

Only one keeper should run per deployment. Two keepers cannot double-charge, because a second successful call on the same billing date reverts, but they could each record a failure for the same missed payment and double the pace of dunning. A merchant who does not want to depend on Levy's keeper can run its own against its own plans, or simply call `charge`.

7.2 Webhooks

Contract events become signed webhooks to the endpoints of the subscription's merchant. There are ten event types, listed in Appendix C, and every payload carries `subId`, `merchant` and the merchant's `ref`. Each delivery carries `X-Levy-Signature`, an Ed25519 signature over `{timestamp}.{deliveryId}.{event}.{body}`, plus the timestamp, delivery id, key id and event type in headers. Receivers verify the signature against the public key the API publishes, reject timestamps more than a few minutes old and drop delivery ids they have seen. Up to 8 deliveries run in parallel with one in flight per endpoint, so each endpoint receives events in order. Failures retry after 1, 5 and 15 minutes, 1 hour and then every 6 hours, and are dead-lettered after 24 hours. A merchant can register up to 5 endpoints.

7.3 API and applications

- **Merchant dashboard.** Monthly recurring revenue normalised to 30 days, active subscriptions, charge success rate, dunning queue, total collected, every subscription with the merchant's reference and allowance runway, webhook management and an invoice export in CSV.
- **Customer portal.** The customer's subscriptions and invoices, pause, resume and cancel from their own wallet, and an allowance top-up when it runs low.
- **Hosted checkout.** `/checkout.html?plan=<id>&ref=<ref>` shows the plan, the merchant, the contract and the customer's balance, asks how many cycles to approve, and states what Levy can pull.

All three are static pages with no third-party scripts, served by the API.

8. Economics

8.1 The fee

Levy charges 1% of each collected payment. There are no tiers, no monthly minimum and no setup fee, and a failed payment costs the merchant nothing. The fee is split out on-chain in the same transaction, so the merchant never invoices Levy and Levy never invoices the merchant.

8.2 Compared with card billing

Monthly price	Card + Stripe Billing	Share	Levy	Share
$10	$0.66	6.6%	$0.10	1.0%
$29	$1.34	4.6%	$0.29	1.0%
$99	$3.86	3.9%	$0.99	1.0%
$499	$18.26	3.7%	$4.99	1.0%

Card figures use Stripe's US list prices on 25 September 2026: 2.9% plus 30 cents per domestic card payment and 0.7% of volume for Stripe Billing [1]. They exclude the 1.5% international surcharge, $15 per dispute, negotiated rates and the services a card processor bundles, such as fraud screening. Levy figures exclude the merchant's cost of converting USDC to another currency, if it needs to.

Two differences do not show in the table. A Levy payment is final: there are no chargebacks, so no dispute fees and no revenue clawed back months later. And settlement happens in the charging transaction, under a second after the call, with no payout schedule.

8.3 Gas

Customers pay gas for approving and subscribing, merchants for creating plans, and the keeper for every charge. We measured execution gas in the contract test suite against a mock USDC token; a transaction adds a base cost of 21,000 gas, and Arc's USDC may cost somewhat more or less than the mock.

Call	Execution gas (measured)	Transaction cost at 20 gwei
`charge`	70,000 median, 113,000 max	about $0.002 to $0.003
`subscribe`, first cycle charged	up to 273,000	about $0.006
`changePlan`	131,000 median, 191,000 max	about $0.003 to $0.004
`createPlan`	161,000	about $0.004
pause, resume, cancel	49,000 to 57,000	about $0.0015

Twenty gwei is Arc's documented minimum base fee, 2×10^{-8} USDC per unit of gas, which matches its target of about $0.001 per ERC-20 transfer [2]. The documented ceiling is 1,000 times higher. Deploying both contracts on Arc testnet cost 0.12 USDC (3.65 million gas at 32.5 gwei).

8.4 Keeper sustainability

The keeper pays for charges out of the protocol fee. A charge earns `price / 100` and costs about $0.002 to $0.003 at the target fee, so a plan pays for its own charges from roughly $0.30 per cycle. A $29 plan earns $0.29 per charge, about a hundred times its gas. Below $0.30 per cycle a plan costs the keeper more than it brings in, which is why a policy that skips such subscriptions is the natural defence against spam (§6.5).

8.5 Where fees go

Fees accumulate in the FeeRouter until anyone calls `sweep()`, which sends the whole balance to the treasury, a Safe multisig on mainnet. The deploy script refuses a mainnet treasury that is not a contract. The FeeRouter is also the seam for any later change to how fees are distributed (§10).

9. Limitations

- **An allowance is not escrow.** A customer can spend their balance or revoke the allowance at any time, and future payments then fail. Merchants get the same assurance a card on file gives: none, plus dunning.
- **No disputes.** Finality protects merchants and removes customers' recourse. A customer who wants money back depends on the merchant's refund policy.
- **Everything is public.** Which wallets subscribe to which merchant, at what price and with which reference, can be read from the chain. Merchant references must never contain personal data.
- **Fixed-length cycles.** A month is a fixed number of seconds. Calendar billing, usage-based billing and currencies other than USDC are not supported.
- **A pause is not the customer's alone** (§4.7).
- **Timeliness depends on a keeper.** Correctness does not, but if no one calls `charge`, merchants are paid late. The keeper runs as a single instance.
- **Off-chain gaps.**
  - Allowance warnings do not read Permit2 allowances, so Permit2 subscribers are flagged low from the start.
  - The event-type header is not covered by the webhook signature (the body identifies the event).
  - Merchants sign in with their payout wallet: there are no scoped API keys and no support for smart-contract wallets (ERC-1271).
  - There is no plan-creation page.
- **Testnet only, and not audited.** The contracts run on Arc testnet, nothing handles real money yet, and everything in this paper has been reviewed by its authors only.

10. Roadmap and future work

The contracts, keeper, API, dashboard, portal and checkout are built, and the contracts have been live on Arc testnet since 26 September 2026. What remains before real money flows:

1. **A soak of at least a week on the testnet deployment,** running real billing cycles end to end: trials, failed payments through to auto-pause, pause and resume, cancellation, plan changes with credit, and the Permit2 path against Arc's own Permit2.
2. **An external audit** of the contracts, with the open items in §6.8 and the exposure bound in §6.2 in scope.
3. **Mainnet deployment** with the treasury multisig in place, and the first real subscription billed.

Planned after launch:

- Carrying each invoice reference on the payment itself through Arc's transaction memos, which are live on mainnet.
- A plan-creation page.
- Scoped merchant API keys.
- Sign-in for smart-contract wallets.
- Reading Permit2 allowances in the runway check.
- A keeper policy for uneconomic subscriptions.

A possible token. One option under consideration for later is a LEVY token, with part of the protocol fee used to buy it back and part distributed to people who stake it. Nothing about it is built or decided. No supply, allocation or launch date exists, and USDC billing works without it. If it goes ahead, it will come only after the protocol carries real fee revenue, and it will be described in a separate document.

11. Conclusion

Levy makes stablecoin subscriptions work the way merchants already bill: one authorisation at checkout, automatic collection every cycle, retries when a payment fails and a clean record of every charge. It does this with the plainest tool available, an ERC-20 allowance, and a contract that turns the allowance into a narrow, predictable permission: one plan price, per billing date, for one merchant. On Arc that permission costs a fraction of a cent to exercise and settles for good in under a second. The design is built, tested and running on Arc testnet. The next steps are a week of real billing on it and an external audit, and this paper will be revised as they happen.

Appendix A. Contract interface

Function	Who can call	Effect
`createPlan(price, cycleSeconds, trialSeconds, graceSeconds, maxFailed)`	Anyone (becomes the merchant)	Creates an immutable plan; emits `PlanCreated`
`subscribe(planId, ref)`	The subscriber	Starts a subscription; charges the first cycle unless the plan has a trial
`subscribeWithPermit2(planId, ref, permit, signature)`	The permit's signer	Grants a Permit2 allowance and subscribes in one transaction
`charge(subId)`	Anyone, once due	Collects one cycle, or records a failure (at most one a day; sooner reverts `RetryTooSoon`)
`changePlan(subId, newPlanId)`	The subscriber	Moves to a same-merchant plan, billed now, less credit
`pauseSubscription(subId)`	Subscriber or merchant	Stops charges until resumed
`resumeSubscription(subId)`	Subscriber or merchant	Fresh start; resets failures
`cancelSubscription(subId)`	Subscriber or merchant	Final
`isChargeable(subId)`	Anyone (view)	Whether the subscription is due for a charge now
`FeeRouter.sweep()`	Anyone	Sends collected fees to the treasury
`FeeRouter.setTreasury(address)`	The treasury	Names a new treasury

Events: `PlanCreated(planId, merchant, price, cycleSeconds)`, `Subscribed(subId, subscriber, planId, nextChargeAt, ref)`, `Charged(subId, amount, fee, invoiceRef, chargeCount)`, `ChargeFailed(subId, failedAttempts)`, `AutoPaused(subId)`, `Paused(subId)`, `Resumed(subId)`, `Cancelled(subId)`, `PlanChanged(subId, oldPlanId, newPlanId, credit)`, and on the FeeRouter `Swept(amount, treasury)` and `TreasuryUpdated(old, new)`.

Appendix B. Parameters

Parameter	Value	Set in
Protocol fee	1% of each pull, rounded down	Contract constant
Cycle length	86,400 s (1 day) to 315,360,000 s (10 years)	Contract constants
`maxFailed`	1 to 255 per plan; auto-pause at twice the value	Plan
Gap between recorded failures	At least 86,400 s (1 day) per subscription	Contract constant
Charge loop / delivery loop	30 s / 5 s	Keeper configuration
Dunning retries	1, 3, then every 7 days after each failure	Keeper configuration
Revert backoff	1 min, 5 min, 15 min, 1 h, then every 6 h	Keeper
Allowance warning	Fewer than 2 rounds left; up to 50 subscribers per cycle; recheck after 6 h	Keeper
Webhooks	5 endpoints per merchant; 5 s timeout; 8 in parallel; retries 1 min to 6 h; dead after 24 h	Keeper and API
Sessions	24 hours	API
Checkout default	12 cycles approved, added to any existing allowance	Checkout

Appendix C. Webhook events

Event	Sent when
`subscription.created`	A customer subscribes
`subscription.updated`	The customer changes plan
`charge.succeeded`	A cycle is collected (amount 0 when credit covered it)
`invoice.ready`	Alongside every `charge.succeeded`, with the same body
`charge.failed`	A charge attempt fails
`dunning.started`	The first failure, with the next retry time

Event	Sent when
`subscription.paused`	Paused by either party, or automatically (`auto: true`)
`subscription.resumed`	Resumed
`subscription.cancelled`	Cancelled
`allowance.low`	The customer's allowance covers fewer than 2 more rounds

Appendix D. Arc testnet deployment

Contract	Address
SubscriptionManager	`0x33c8e7e46583bdfd2e001f55ac1c58bd84eb739f`
FeeRouter	`0xa6efee5f5a2eb2afae102f790b4889fcc01fe848`

Arc testnet (chain 5042002), deployed on 26 September 2026 in block 64031023. The runtime bytecode of both contracts matches a build of the contract sources at commit `08dd7b7`, apart from the addresses set at construction. On testnet the fee treasury is the deploying wallet; on mainnet it will be a Safe multisig. An earlier deployment the same day, in block 64020204, predates the fix in §6.8 and is not used.

References

1. Stripe, "Pricing & Fees", https://stripe.com/pricing (accessed 25 September 2026).
2. Arc Docs, "Gas and fees", https://docs.arc.io/arc/references/gas-and-fees (accessed 25 September 2026).
3. Arc Docs, "Deterministic finality and settlement", https://docs.arc.io/arc/concepts/deterministic-finality (accessed 25 September 2026).
4. Arc Docs, "Contract addresses", https://docs.arc.io/arc/references/contract-addresses (accessed 25 September 2026).
5. EIP-20, "Token Standard", https://eips.ethereum.org/EIPS/eip-20.
6. Uniswap, "Permit2 overview", https://developers.uniswap.org/docs/protocols/permit2/overview.
7. ERC-1337, "Subscriptions on the blockchain" (2018, stagnant), https://eips.ethereum.org/EIPS/eip-1337.
8. EIP-948 discussion, "Recurring Subscription Models", https://github.com/ethereum/EIPs/issues/948.
9. EIP-712, "Typed structured data hashing and signing", https://eips.ethereum.org/EIPS/eip-712.
10. EIP-4361, "Sign-In with Ethereum", https://eips.ethereum.org/EIPS/eip-4361.
11. RFC 8032, "Edwards-Curve Digital Signature Algorithm (EdDSA)", https://www.rfc-editor.org/rfc/rfc8032.
12. EIP-1559, "Fee market change for ETH 1.0 chain", https://eips.ethereum.org/EIPS/eip-1559.

This paper describes software. It is not an offer or solicitation to buy any token or other asset, and it promises no future feature, including the token mentioned in §10. The contracts are on Arc testnet only and not yet audited. Gas figures come from local tests and may differ on Arc.