Fees on Aztec may be settled in four primary ways from the user's POV:

1. Fee Juice

: A potential protocol-enshrined fee token, similar to ETH on Ethereum or STRK on Starknet. From the protocol's perspective, this may be the only way to pay fees. Fee juice is a public contract. Therefore, the fee juice balance of an address is public and transaction costs (in fee juice) are also public. Additionally, fee juice is non-transferable. Fee juice is only obtainable by bridging from L1. This bridge is one-way (i.e. a user can deposit into Aztec but can't withdraw back).

1. Fee Payment Contracts (FPCs)

: These contracts act as the fee payer on behalf of the user. They allow users to pay fees in various tokens, with the contract exchanging them for fee juice. FPCs offer enhanced flexibility; fee payments (and refunds) may be private and/or public. It is even possible for FPCs to decide to use additive homomorphic encryption to allow sequencers to process fee refunds without revealing the user identity or the token they paid fees in.

1. L1 Fee Coverage

: Users can directly pay L2 transaction fees from their L1 fee juice balance.

1. App-sponsored transactions

: Apps can pay for user's transactions.

The flexibility and choice of privacy offered to the users make FPCs extremely attractive and it also may a potential revenue source for the FPC operator.

Some user stories:

To better understand fees and FPCs, let's define some actors:

- Alice

: A user sending a transaction.

- Fee Juice

: The protocol-defined method for paying fees.

- USDP

: A random token used as an example for private USD payments.

- FPC

: A fee payment contract accepting USDP.

- App

: An application that a user interacts with.

Scenario 1 - Alice pays publicly in fee juice

Assume Alice has already bridged Fee Juice from L1 to her L2 account.

1. Transaction Preparation

: Alice constructs a transaction payload specifying the desired app interaction, gas limits, maximum fee amount, and fee payment method (Fee Juice in this case).

1. Fee Balance Check

: Alice's entrypoint enqueues a public function fee_juice.check_balance(max_fee)

to verify sufficient Fee Juice balance.

1. Private Computation

: execute private app logic, which may include enqueuing public function calls.

1. Fee Payment

: Enqueues another public function fee_juice.pay_fee(max_fee)

for the sequencer to deduct the fee at transaction completion.

1. Now the transaction is sent to the sequencer:
2. The fee_juice.check_balance

function is executed. If insufficient funds, the transaction reverts.

- The sequencer executes any public functions related to the app logic
- The fee_juice.pay_fee

function is executed, deducting the specified fee amount from Alice's Fee Juice balance.

1. The fee_juice.check_balance

function is executed. If insufficient funds, the transaction reverts.

1. The sequencer executes any public functions related to the app logic
2. The fee_juice.pay_fee

function is executed, deducting the specified fee amount from Alice's Fee Juice balance.

Note: as mentioned before, fee juice is public so paying fees directly with fee juice is a huge privacy leak since it reveals the address which did a transaction and the transaction itself may be guessable based on how much fee juice was consumed.

[

|2047.2102430877712x1428.656743240882

1600×1116 107 KB

](https://europe1.discourse-cdn.com/flex013/uploads/aztec/original/2X/a/a78c53a493e23d01e37ab44ea289ba9639b0dfc7.png)

Scenario 2 - Alice pays publicly in USDP via FPC

Assume the FPC is already funded with Fee Juice (from the L1 to L2 bridge).

1. Transaction Preparation

: Alice constructs a transaction payload specifying the desired app interaction, gas limits, maximum fee amount (in Fee Juice), and indicating payment via the USDP FPC.

1. Public approval

: Alice approves FPC to do a USDP.transfer_public

on her behalf for an amount equal to the maximum fee plus a commission.

1. FPC Public setup

: She then privately calls FPC.public_fee_entrypoint()

which sets FPC as the fee payer and enqueues a public call to transfer USDP to the FPC. This function also enqueues a public "teardown" function to be executed later for fee payment in Fee Juice and potential USDP refunds.

1. Sequencer Processing

: The sequencer picks the transaction and executes public functions * The FPC's setup phase is executed. If Alice lacks sufficient USDP, the approval is incorrect, or the FPC is out of Fee Juice, the transaction reverts.

- The sequencer then executes the public app logic.
- The FPC's teardown function is executed, processing any refunds to Alice in USDP.
- The FPC's setup phase is executed. If Alice lacks sufficient USDP, the approval is incorrect, or the FPC is out of Fee Juice, the transaction reverts.
- The sequencer then executes the public app logic.
- The FPC's teardown function is executed, processing any refunds to Alice in USDP.

[

1600×1136 108 KB

](https://europe1.discourse-cdn.com/flex013/uploads/aztec/original/2X/4/4d15d38403af2d16b3ca130ff491a7fe2f894a0a.png)

Some Callouts:

Scenario 3: Alice pays privately in USDP via FPC

Assuming the FPC is already funded with Fee Juice:

1. Transaction Preparation

: Alice prepares a transaction specifying the target application, fee payment method (USDP FPC), gas limits, maximum USDP fee, and a random value for refund note generation.

1. Private Approval

: Alice creates a private approval (auth witness) for FPC to consume Alice's funds.

1. FPC Private Setup

: Alice privately calls FPC's fund_transaction_privately()

which sets FPC as the fee payer and also calls USDP.setup_refund()

.

1. USDP Refund Setup

: The USDP contract verifies the approval, consumes the maximum specified USDP amount from Alice's balance, and generates two partial token notes (one for the potential refund to Alice, the other for the FPC's fee). A public call to USDP.complete_refund

is enqueued to finalize these notes later.

1. Sequencer Processing

: The sequencer processes the transaction, executing public functions. The FPC is charged the equivalent fee in Fee Juice. The USDP.complete_refund function is executed, finalizing the token notes with the exact refund amount and FPC fee.

1. Note Redemption

: Alice and the FPC can later manually add the completed notes to their respective balances.

Scenario 4: Alice receives her refunds privately too via the FPC

This builds upon the previous scenario where Alice paid fees privately in USDP via an FPC. Here she can receive a refund from the payment. Keeping this as a separate scenario sets the mental model for an FPC that may optionally choose not to process refunds to decrease Alice's proving time (since all private functions are circuits which must be proven locally) or save on complexity.

1. FPC Private Setup

: Alice privately calls FPC's fund_transaction_privately()

which sets FPC as the fee payer and also calls USDP.setup_refund()

.

1. USDP Refund Setup

: The USDP contract verifies the approval, consumes the maximum specified USDP amount from Alice's balance, and generates two partial token notes (one for the potential refund to Alice, the other for the FPC's fee). A public call to USDP.complete_refund

is enqueued to finalize these notes later.

1. Sequencer Processing

: The sequencer processes the transaction, executing public functions. The FPC is charged the equivalent fee in Fee Juice.

The USDP.complete_refund

function is executed, finalizing the token notes with the exact refund amount and FPC fee.

1. Note Redemption

: Alice and the FPC can later manually add the completed notes to their respective balances.

[

1600×816 84.8 KB

](https://europe1.discourse-cdn.com/flex013/uploads/aztec/original/2X/b/b389a254a56f7c59a26276bc0355ca53d9e53643.png)

Example Code

FPC with public payments

FPC with private payments and refunds

End-to-end test using aztec.js (including a scenario for app sponsoring transactions!)

Additional Reading

For a deeper dive into fee mechanics, including setup and teardown phases, please refer to the first two sections of our protocol specifications.

To explore the security implications associated with Fee Payment Contracts (FPCs), consult Appendix 1

Shoutout to @alexghr @mitch and @Jan for their prior work on this.

Appendix 1: FPC Security and Caveats

Exchange Rate Dynamics and FPC griefing

- FPCs require a consistent supply of Fee Juice, bridged from L1, to function effectively.
- For simplicity, above diagrams and current tests and example contracts on our github assume a fixed 1:1 exchange rate between Fee Juice and USDP. This assumption may even work for testnet but is unrealistic for mainnet operations.
- FPCs should function as market makers, set exchange rates while minimizing slippage to avoid overcharging users. Competition among FPCs is expected to drive down spreads. Our end-to-end tests are based on this assumption.
- Accurate fee estimation is crucial to prevent overcharging users or FPC being grifed i.e. incurring unexpected costs (such as teardown function becoming pricey).

Public Setup and DoS

FPCs with public setup phases (Scenario 2) pose a potential DoS risk to the sequencer, as reverts during the public setup also incur fees. To mitigate this, we are exploring options such as sequencer whitelisting for FPCs or simulating transactions to predict potential reverts.

Teardown functions

FPCs with private setup (Scenario 3/4) must define a function to be called in the “teardown” phase (which usually processes refunds of tokens).

- To facilitate, the transaction fee (via context.tx_fee) is known when teardown runs
- Teardown function is then a special function with its own separate gas limit (teardown_gas_limit) that the user must pre-pay for. This is shown in the GasSetting class.
- There are no refunds for the teardown gas (to prevent an endless spiral of payments and refunds)
- FPCs must accurately estimate the required Fee Juice for both. This means both teardown and setup should be minimal and standard. This is similar to EVM based Paymasters which need to prevent themselves from being griefed whilst being competitive. As a tangent: Typical L2 sequencers also face a similar issue where they must estimate L1 congestion ahead of time so as to charge enough gas on L2 but also not overcharge users!

- If the transaction attempts to overwrite the teardown function, proof generation will fail (otherwise this could let you could call the refund function multiple times!).

DISCLAIMER

The information set out herein is for discussion purposes only and does not represent any binding indication or commitment by Aztec Labs and its employees to take any action whatsoever, including relating to the structure and/or any potential operation of the Aztec protocol or the protocol roadmap. In particular: (i) nothing in these posts is intended to create any contractual or other form of legal relationship with Aztec Labs or third parties who engage with such posts (including, without limitation, by submitting a proposal or responding to posts), (ii) by engaging with any post, the relevant persons are consenting to Aztec Labs' use and publication of such engagement and related information on an open-source basis (and agree that Aztec Labs will not treat such engagement and related information as confidential), and (iii) Aztec Labs is not under any duty to consider any or all engagements, and that consideration of such engagements and any decision to award grants or other rewards for any such engagement is entirely at Aztec Labs' sole discretion. Please do not rely on any information on this forum for any purpose - the development, release, and timing of any products, features or functionality remains subject to change and is currently entirely hypothetical. Nothing on this forum should be treated as an offer to sell any security or any other asset by Aztec Labs or its affiliates, and you should not rely on any forum posts or content for advice of any kind, including legal, investment, financial, tax or other professional advice.