FlashExecutor — Integrator Guide

Version 2.2 · 19 September 2026 Cradle Capital Corp · Atomic Flash Desk

VERSION	DATE	CHANGE
2.2	2026-09-19	**BNB Chain (56) added to §2.4 and §8.** It has been served and signable throughout, and both tables omitted it — the deployment list said "all three chains" while the desk served four. Also records a service defect fixed the same day: the authorisation endpoint resolved its executor from process configuration rather than from the chain table, so it signed under the **v1** scheme (v1 typehash, EIP-712 domain version "1", v1 verifying contract, no `positionsHash`) while `/healthz` reported v4. A v4 digest reproduced from this guide could not match one issued before that fix. Nothing in the documented formula changed; the service now matches it.
2.1	2026-08-22	Copy fix in §2.3: the refused-target table read "interfere with the loan servicing you", where the trailing "you" left the clause dangling. It now names the loan meant — "the loan servicing this flash". No behavioural change.
2.0	2026-08-14	**Rewritten against Executor v4, now deployed on all three chains.** The authorisation digest gains a tenth field, `positionsHash`; `execute()` gains a `positions` argument; `Action` gains `balanceToken` and `dataOffset`. Every one of those changes the digest, so **a v1-era integration will not verify** — see §4. Adds balance reads (§2.1), position builds (§2.2), multi-source drawings (§6) and a deployment table (§8). Funding sources are now described by economics rather than by name.
1.5	2026-08-06	The true-up is withdrawn (Fee Schedule v2.0). §3.3 rewritten: no credit, no rebate, no cash on closure — the tier reached becomes the following month's opening rate, and a closed month is final. No contract change.
1.4	2026-08-06	Base is live. §7 replaced: the guide had said "one chain today".
1.3	2026-08-05	Adds the base URL. The endpoint had been documented as a bare path with no host.
1.2	2026-08-05	`feeBps` becomes variable within a month. Documents the volume ratchet and the consequence integrators must handle: a fixed `minProfit` changes meaning as the rate moves.
1.1	2026-08-04	Corrects an understatement: action targets are unrestricted bar three. v1.0 read as though the venue allowlist bound strategies; it binds only the flash source.
1.0	2026-08-04	First issue.

Read this first if you integrated before 14 August 2026

Executor v4 is deployed on Ethereum, Base and Polygon. **Its authorisation digest is not the one earlier versions of this guide describe.** Three changes each alter the signed message:

- `Authorization` now carries a tenth field, `positionsHash`
- `Action` now carries `balanceToken` and `dataOffset`, both inside the action hash
- `PositionCheck` now carries `owner`, inside the position hash

An integration that reproduces the old nine-field digest will compute a different value, and the contract will reject the signature with `BadAuthorization`. **It fails closed** — nothing is lost beyond gas, and no drawing executes on a stale digest. But nothing works either, so reproduce the digest against §4 before your next live drawing.

If you do not reproduce the digest yourself and simply pass through what the desk returns, no change is required of you.

What this is

You bring the strategy. We provide flash liquidity and take a share of the profit, on-chain, inside the same transaction. If the strategy does not profit, the transaction reverts and nothing has happened except your gas.

You execute. We authorise. Every drawing needs a signature from us over your exact parameters, and only you can use it. We never send the transaction, never hold your capital, and cannot move it.

1. The shape of a drawing

One call, one transaction, four phases:

1. `execute()` takes a flash loan of `amount` in `asset` from an allowlisted source.
2. Your `Action[]` runs in order — whatever your strategy does.
3. The contract checks it can repay, and that any positions you declared were actually built. **If**

either fails, everything reverts.

1. The fee goes to `feeSink`; the remainder goes to you; the loan is repaid.

You never receive the borrowed principal in your own account. It is borrowed, used and repaid inside one atomic call.

2. The call

```
function execute(
    address pool,                       // an allowlisted liquidity source; the desk tells you which
    address asset,                      // what you are borrowing
    uint256 amount,                     // how much
    Action[] calldata actions,          // your strategy, in order
    PositionCheck[] calldata positions,  // what the drawing must have BUILT (may be empty)
    address[] calldata sweepTokens,     // any other token your actions may leave behind
    uint16  feeBps,                     // our cut of GROSS PROFIT, from the authorisation
    uint256 minProfit,                  // your floor, after our fee
    uint256 deadline,                   // authorisation expiry
    uint256 nonce,                      // authorisation uniqueness
    bytes   calldata signature          // our EIP-712 signature over exactly the above
) external returns (uint256 netProfit);
```

`positions` is the v4 addition and it selects the settlement mode. Empty means the arbitrage path: the drawing must end with more of `asset` than it started with. Non-empty means a position build — see §2.2.

The Action struct

```
struct Action {
    address target;         // the contract to call
    address approve;        // optional: token to approve to `target` first (address(0) to skip)
    uint256 approveAmount;  // exact amount; never infinite
    uint256 value;          // native value to forward
    address balanceToken;   // address(0) = use `data` verbatim; otherwise see §2.1
    uint256 dataOffset;     // byte offset of the word to overwrite with the live balance
    bytes   data;           // calldata — arbitrary by design
}
```

`data` is arbitrary. That is deliberate: your strategy is yours, and we do not want to know it in advance or constrain it to a menu.

2.1 Balance reads — the sizing buffer is gone

This is the most valuable change in v4 and the one most likely to improve your realised numbers.

`execute` runs fixed calldata. Under v1 that meant every leg's input had to be a literal committed before the previous leg revealed what it actually produced. Legs were therefore sized *under* their quote and the shortfall swept back — safe, and expensive. Measured on our own Ethereum book: quoted −1.20 bps, executable −11.09 bps. **That ~100 bps buffer was larger than any edge we found in 1,215 loops.**

Set `balanceToken` and the Executor reads its own balance of that token at the moment the action runs, and splices it into `data` at `dataOffset`, replacing a placeholder word. The leg then spends exactly what the leg before it produced. No buffer, no residual, no guess.

```
Action 2: balanceToken = WETH
          dataOffset   = 4 + 32          // the amountIn word of a swap, past the selector
          data         = swapExactInput(<placeholder>, …)
```

`dataOffset` is the byte position of the 32-byte word to overwrite. It must clear the 4-byte selector and sit wholly inside the calldata; both are checked, and a bad offset reverts with `BadDataOffset(index, offset, dataLength)`. That check exists because an offset which silently overwrote the wrong word would corrupt a swap into something you never authorised.

Both fields are inside the action hash, so a signature cannot be executed with a different token read or a different offset.

2.2 Position builds — when the drawing ends holding an asset, not a profit

The arbitrage path requires the drawing to end with more of `asset` than it began with. A great deal of useful work does not: building a levered position, swapping collateral, refinancing a borrow from one venue to another, de-levering, or self-liquidating all end holding a **position**, and would be rejected by a profit test.

`PositionCheck` is how you declare that:

```
struct PositionCheck {
    address token;      // the receipt or collateral token the drawing must have produced
    uint256 minAmount;  // the least of it you will accept
    address owner;      // whose balance is measured; address(0) means the caller
}
```

If the balance of `token` held by `owner` has not risen by at least `minAmount` across the drawing, it reverts with `PositionNotBuilt(token, have, required)`. The check is the position build's equivalent of `minProfit`: it is your protection against a build that half-executed.

`owner` is the v4 addition, and it exists so that gas and ownership can be different addresses. Earlier versions measured what the *caller* received, and the caller is `msg.sender`. When a desk-held key sends a build on a trader's behalf, the receipt tokens are credited to the trader, so the caller's balance never moves and the guard failed a drawing that had worked perfectly. Naming the owner lets a trader stop signing every drawing without giving up the only on-chain proof that the drawing built what it promised. `address(0)` reproduces the old behaviour exactly, so an existing integration composes identically.

`owner` is inside the position hash. A signature that did not cover it could be replayed to credit the check against somebody else's balance.

2.3 The venue list is not a menu — read this before assuming you are constrained

Three targets are refused, always. Everything else is permitted:

REFUSED TARGET	WHY
the lending pool	would let you interfere with the loan servicing this flash
the executor itself	re-entrancy
`address(0)`	not a target

Any of these reverts with `TargetNotAllowed(target)`.

Those three refusals are the *entire* restriction. There is no venue allowlist on your actions, and this is the single most misread thing about the contract:

> **Where the money comes FROM is constrained. What you DO with it is not.**

So a strategy that borrows from us and then trades on **Compound, Sky, Liquity, Morpho, Curve, Balancer, any DEX or any protocol we have never heard of** is supported today, on the deployed contract, with no change and no permission from us.

Concretely, all of these are ordinary uses:

- flash a stablecoin at **0 bps**, liquidate a position on a lending venue, repay
- flash, arb between a curve-style pool and a concentrated-liquidity tier, repay
- flash, unwind a trove, sell the collateral, repay
- flash, build a levered position and declare it via `positions` (§2.2)
- anything else that returns more than `amount + premium` inside one transaction, or builds what it

said it would

We do not need to add your venue. We do not need to know your venue. If your strategy fits in an `Action[]` and repays the loan, it works.

Approvals are set to the **exact** amount and zeroed after. A leftover allowance reverts the whole call with `DanglingAllowance` — the contract is stateless between transactions and enforces it.

2.4 Which source, and what it costs

The desk gives you the source address for your facility, and `POST /v1/desk/authorize` returns it with the authorisation. We select the cheapest source that can cover your size, and we describe our funding by economics rather than by name — the same reason a broker does not publish its liquidity providers.

What we will tell you, because it is what you are actually paying:

CHAIN	SOURCES REACHABLE	PREMIUM RANGE	DEEPEST SINGLE SOURCE
Ethereum	3	0–5 bps, two at **0**	~$110M+
Base	4	0–5 bps, two at **0**	**~$201M at 0 bps**
BNB Chain	2	0–5 bps, one at **0**	read it live
Polygon	1	5 bps	thinnest of the four

Premiums are public on chain and governance-settable, so we read them live rather than carrying a constant — `FLASHLOAN_PREMIUM_TOTAL()` on the source address answers, and you should not take our word for it. On a $1,000,000 drawing the difference between a 0 bps and a 5 bps source is $500 of cost the chain does not require you to pay, which is why routing is ours to get right.

What protects you is atomicity, not opacity. `isPool` is a public mapping, you supply `pool` in your own transaction, and the trace shows which source serviced the loan. If the loan is not repaid, the whole transaction reverts, and that holds whoever funded it.

---

3. The fee, and what happens when a trade loses

Inside the flash callback, after your actions have run:

```
uint256 owed  = amount + premium;              // principal + the source's own fee
if (have <= owed) revert NotProfitable(have, owed);
uint256 gross = have − owed;
uint256 fee   = (gross ∗ feeBps) / 10_000;
uint256 net   = gross − fee;
if (net < minProfit) revert BelowMinProfit(net, minProfit);
```

Read this carefully, because it is the whole commercial arrangement:

- **The fee is a share of gross profit**, computed only after the loan and the source's premium are

covered. It is never a slice of notional.

- **An unprofitable strategy reverts.** You pay gas; nothing moves; we earn nothing. We cannot lose

money on your trade and neither can you beyond the gas.

- **`minProfit` is yours**, applied after our fee. Set it to the least you would accept. If the fill

is thinner than that, the transaction reverts rather than settling.

- `feeBps` is fixed in the authorisation we sign. It can never exceed `maxFeeBps`, which is set at

deployment and immutable, and which the constructor itself caps at 5000.

The fee transfer happens **before** you can reach your profit. It cannot be routed around.

3.1 `feeBps` is not constant — it moves through the month

This is the one thing in this guide most likely to surprise an automated integrator.

`feeBps` is stamped per authorisation from your **month-to-date drawn volume**, and it ratchets down as you cross each threshold:

VOLUME DRAWN SO FAR THIS MONTH	`FEEBPS`	OUR SHARE
Up to $1M	4000	40%
$1M – $10M	3000	30%
$10M – $100M	2000	20%
Over $100M	1000	10%

The counter resets at 00:00 UTC on the 1st. Two authorisations for identical parameters, issued a week apart, can legitimately carry different `feeBps`.

What this means for your code:

- **Never hardcode or cache `feeBps`.** Use the value returned with each authorisation. If you

reproduce the digest yourself (§4), reproducing it with a remembered rate yields a different digest and `BadAuthorization` at execution — after you have paid gas.

- **Do not derive `feeBps` from the schedule yourself.** Your view of your own volume and ours can

differ over reverted transactions and mid-flight draws. Ours is the one inside the signature.

- **The contract cap is unchanged.** `maxFeeBps` is immutable; no authorisation we issue can exceed

it, whatever the schedule says.

A draw is priced **inclusive of itself**: at $900k month-to-date, a $5M request is authorised at 3000, not 4000.

3.2 A fixed `minProfit` does not mean a fixed threshold

`minProfit` is tested against profit **net of our fee**. So the gross a trade must produce to clear a given floor is `minProfit / (1 − rate)` — and since the rate moves, so does that requirement:

`FEEBPS`	GROSS NEEDED PER UNIT OF `MINPROFIT`	ON `MINPROFIT` = $10,000
4000	1.6667×	$16,666.67
3000	1.4286×	$14,285.71
2000	1.2500×	$12,500.00
1000	1.1111×	$11,111.11

The first tier demands 50% more gross profit than the last for the same floor. A trade grossing $15,000 against a $10,000 floor:

`FEEBPS`	NET AFTER FEE	RESULT
4000	$9,000	**reverts** `BelowMinProfit`
3000	$10,500	settles
2000	$12,000	settles
1000	$13,500	settles

The same trade, at the same size, on the same venue, reverts in the first days of a month and settles later in it. If you carry a constant `minProfit`, that is what you will see, and it will look like a venue or liquidity problem rather than a fee one.

Derive `minProfit` from the `feeBps` in the authorisation, not from a constant.

3.3 Your rate carries across the month boundary — nothing else does

A fee that has settled cannot be returned on-chain: the contract is ownerless and has no rescue function. **The desk does not try to.** There is no credit, no rebate and no balance. The tier your volume reaches becomes the *opening* `feeBps` of the authorisations we issue you next month (Fee Schedule v2.0, section 11).

Two practical consequences:

1. Do not reconcile a single transaction's fee against the headline rate. A month that ends over $100M does not pay 1000 bps on every transaction — it pays 4000, then 3000, then 2000, then 1000 as it climbs. On-chain, `Executed(caller, asset, amount, gross, fee, …)` always reports the fee actually taken. Reconcile against the month, not the transaction.

2. Do not carry a receivable for the difference. Fee Schedule v1.2 and earlier promised a true-up credit against the following month, with cash settlement on closure. **That promise is withdrawn as of v2.0.** A closed month is final. If your accounting accrued a credit under the old schedule, write it off rather than carrying it.

Your first month costs the blend; every month after costs the headline rate, provided you keep drawing. The grace lasts one month and derives from the previous month's own volume, so a dormant month lets the opening rate decay back toward 4000.

---

4. Getting an authorisation

The easy path: ask the desk.

```
POST https://desk-api.atomicexchange.cloud/v1/desk/authorize
```

Base URL: `https://desk-api.atomicexchange.cloud` — every path below is relative to it. Send your parameters; we return the signature, the nonce, the deadline, the source address **and** `feeBps`. You do not build a digest, track nonces, reproduce any hashing, or work out your own tier.

The endpoint enforces your facility before signing — accepted application, accepted Onboarding Schedule, limit, maximum notional per transaction, and maximum concurrent authorisations — so a request that would breach your terms is refused with a reason rather than signed and reverted on chain at your expense.

`feeBps` in the response is authoritative and must be passed to `execute()` verbatim. Substituting any other value changes the digest and fails verification.

Two supporting routes:

- `GET /healthz` — whether the desk can sign right now, per chain, including how current its draw

index is. It reports the running version and every executor address it will sign against. Check it before concluding a change is or is not live.

- `GET /v1/desk/networks` — the chains we serve, the assets we can price on each, the executor

address, and the capability flags (`twoLeg`, `threeLeg`, `positionBuild`, `stackedSources`).

If the Executor is not deployed on the chain you asked for, or no signer key is configured, you get a 503 explaining that an authorisation issued in that state would fail verification and cost you gas for nothing — not a signature that does not work.

The independent path — reproducing the digest

For funds that would rather verify our signature themselves before spending gas. Both paths produce the same authorisation; this one just means you trust arithmetic you did yourself.

```
domain: EIP712Domain(string name, string version, uint256 chainId, address verifyingContract)
        name    = "AtomicFlashExecutor"
        version = "1"

Authorization(address caller,address asset,uint256 amount,uint16 feeBps,uint256 minProfit,
              uint256 deadline,uint256 nonce,bytes32 actionsHash,bytes32 sweepTokensHash,
              bytes32 positionsHash)
```

`positionsHash` is the tenth field and is new in v4. It is signed because the positions array *selects the settlement mode*. Without it, an authorisation signed for a position build could be submitted with `positions = []` and routed into the arbitrage path, where gross is zero for a build and the fee is never charged. The signature verified perfectly. Anything that changes what a drawing does belongs in the digest.

The two sub-hashes are `keccak256` over packed encodings, in array order:

```
actionsHash   = keccak256( ‖ᵢ  target ‖ approve ‖ approveAmount ‖ value
                             ‖ balanceToken ‖ dataOffset ‖ keccak256(data) )

positionsHash = keccak256( ‖ᵢ  token ‖ minAmount ‖ owner )

sweepTokensHash = keccak256( ‖ᵢ  token )
```

`balanceToken`, `dataOffset` and `owner` are all inside these hashes. Each was added to a struct before it was added to the corresponding hash, and in each case the gap was exploitable — a valid authorisation could be executed with a different token read, a different calldata offset, or a position credited to a different account. If you are reproducing these, include them.

You can check your work against the contract before spending gas:

```
function authorizationDigest(
    address caller, address asset, uint256 amount, uint16 feeBps, uint256 minProfit,
    uint256 deadline, uint256 nonce, bytes32 actionsHash, bytes32 sweepTokensHash,
    bytes32 positionsHash
) external view returns (bytes32);
```

Ten parameters. The v1 helper took nine; calling the old shape will not compile against the deployed contract, which is the failure we want.

The authorisation is bound to you

`caller` is inside the struct hash, and `execute()` builds the digest from `msg.sender`. **An authorisation issued to you is unusable by anyone else** — a different caller produces a different digest and the signature fails. Nobody can front-run your authorisation, including us.

Each digest is single-use: `authorizationUsed[digest]` is set on first use, and a replay reverts with `AuthorizationUsed`. Because `caller` is in the digest, two integrators can use the same `nonce` without colliding.

---

5. Checking that a signature will be accepted

Do not check `isSigner` alone. It records membership, which is historical fact. Authority is:

```
isSigner[a] && !signerRevoked[a]
```

`isSigner` stays `true` after revocation, deliberately — collapsing the two would make a revoked address look as though it had never been admitted. **Read both.** An integrator who checks only the first will believe a dead authorisation is live.

The signer set is **1-of-n, fixed at construction, and can only shrink.** Membership can be revoked, one way, and never granted. A set that could grow would make a compromised key permanent; a single immutable key would make rotation a redeployment.

Verification happens at **execution** time, not signing time. An authorisation signed before a revocation, still inside its deadline, dies with the key. That is what makes revocation an incident response rather than a formality.

---

6. Multi-source drawings

v4 can draw on **more than one source inside a single transaction**, including two sources of the *same* asset. A nested loan is triggered by one of your actions calling a second source directly; the Executor accumulates what is owed per asset and settles all of it before the outermost frame closes.

- Maximum nesting depth is **4**. Beyond it, `NestTooDeep(depth)`.
- Same-asset stacking is supported. v1 forbade it outright; v4 accumulates per-asset debt instead,

which makes it safe rather than forbidden.

- **Availability differs by chain.** Ethereum and Base support it; Polygon has a single source, so

there capacity is that source's depth and never a sum. `GET /v1/desk/networks` reports `stackedSources` per chain — read it rather than assuming.

Use it when your size exceeds the deepest single source, or when a cheaper shallow source can cover part of the draw. The desk composes this for you if you ask it to.

---

7. Failure vocabulary

Every revert is a named custom error. The ones you will meet:

ERROR	MEANING
`NotProfitable(have, owed)`	your strategy did not cover principal + premium
`BelowMinProfit(profit, required)`	it profited, but under your own floor
`PositionNotBuilt(token, have, required)`	a declared position was not built to `minAmount`
`CannotRepay(have, owed)`	the drawing could not return what it owed a source
`BadAuthorization()`	signature invalid, or signer revoked — **and the most likely cause of a v1-era integration failing against v4 (see §4)**
`AuthorizationUsed(digest)`	this authorisation has already been spent
`Expired()`	past `deadline`
`FeeTooHigh(bps)`	`feeBps` above `maxFeeBps`
`ActionFailed(index, reason)`	action `index` reverted; `reason` is its raw revert data
`BadDataOffset(index, offset, dataLength)`	a balance read pointed outside the calldata or into the selector
`TargetNotAllowed(target)`	the source, the executor itself, or the zero address
`PoolNotAllowed(pool)`	not an allowlisted source
`PoolKindMismatch` / `NotAPool`	the source was called with the wrong shape for its type
`NestTooDeep(depth)`	more than four nested drawings
`ResidualBalance(token, amount)`	a token was left behind — add it to `sweepTokens`
`DanglingAllowance(token, spender, amount)`	an approval survived the call
`Reentrancy()`	`execute()` was re-entered
`ZeroAmount` / `NoActions`	empty drawing

`ActionFailed` carries the index and the raw revert data of your own call, so a failure inside your strategy is attributable to the step that caused it.

8. Deployments

Executor v4, live on all four chains.

```
Ethereum    chainId 1      0x2323C5f1707ce385B9A6FBA4610984D284D231B1
Base        chainId 8453   0x1820B0CC9BC92FBE5dea958bFBD5E29df6BD257E
BNB Chain   chainId 56     0x675Dc4c92EFCaE6bF0a11d697b90F4aDEcB82F70
Polygon     chainId 137    0xdAac3D7cA20f74220dCff0C2ccf2163bA49bEc60
```

CHAIN	SOURCES	MULTI-SOURCE DRAWINGS
Ethereum	3, two at 0 bps	yes
Base	4, two at 0 bps	yes
BNB Chain	2, one at 0 bps	yes
Polygon	1, at 5 bps	no

All four support two-leg, three-leg and position-build drawings, and all four carry the same `maxFeeBps` ceiling. `GET /healthz` is the authority on this list: it reports every executor the desk will sign against, per chain, with its version — check it rather than this table if the two ever disagree.

Earlier executor versions remain deployed and are **not** retired — they hold no state, so an integration pinned to one keeps working. They do not receive new features, and the desk signs for whichever address it is pointed at. `verifyingContract` is inside the EIP-712 domain, so a signature for one executor cannot be used against another.

`GET /v1/desk/networks` is the authoritative list, including the assets priceable on each chain. Treat this table as a convenience and that endpoint as the source of truth.

---

9. What we can and cannot do

We cannot:

- move your funds, or touch them at any point — the executor holds nothing between transactions
- execute on your behalf, or send any transaction for you
- use an authorisation we issued to you
- change the fee ceiling, the fee recipient, or the sources, after deployment — all immutable
- add a signer, ever

We can:

- decline to authorise
- revoke a signing key, which invalidates in-flight authorisations from that key

Nobody can, including us: upgrade the contract, pause it, or withdraw from it. There is no owner, no admin, no proxy and no rescue function.

---

10. Before you integrate

1. **Reproduce `authorizationDigest` in your own tooling and check it matches ours for a known**

input. **Do this before your first live drawing. If you integrated before 14 August 2026, do it before your *next* one — see the notice at the top of this guide.**

1. **Use balance reads on every multi-leg drawing** (§2.1). If you are still sizing legs under a

quote and sweeping the residual, you are paying a buffer that was measured at roughly 100 bps and is no longer necessary.

1. **Simulate against a fork at your real clip size.** A spread that exists at $1,000 frequently does

not at $100,000 — impact is not linear, and top-of-book is a number at zero size.

1. **Set `minProfit` deliberately.** It is your only protection against a thin fill, and it is applied

after our fee. Derive it from the authorisation's `feeBps`, not from a constant (§3.2).

1. **Declare positions if your drawing builds one** (§2.2), and set `owner` if the account paying gas is not the account receiving the position.
1. **List every token your actions can leave behind in `sweepTokens`.** Anything unlisted and non-zero reverts the drawing.
1. **Agree your facility terms with us** — limit, maximum notional per transaction, maximum concurrent authorisations, and fee. These are set at underwriting, per counterparty.

Questions to the desk. This document describes the contract as deployed; where it and the source disagree, the source is authoritative.