ChatGPT

Current Whitelist Generation: Go vs. Python Approaches

Our project has ended up with **multiple implementations of the Idena identity whitelist logic**, which is causing duplication and potential inconsistencies. In particular, we have:

- **Go-based "rolling indexer" logic (in `IdenaAuthGo`)** – primarily in `main.go` (with functions like `buildEpochWhitelist`) and a background agent (`identity_fetcher`). This uses the local Idena node's RPC (e.g. `dna_epochIdentities`) and some REST calls to gather identities.
- **Python prototype (`whitelist_blueprint`)** – a script that uses the **official Idena HTTP API** to fetch all relevant identities and filter them [1] [2]. This was our reference implementation for correctness.

These two implementations perform *the same fundamental steps*: find all identities who participated in the last validation, then filter out those not meeting the criteria (state, stake, no bad flips). Unfortunately, **we've been maintaining both**. The Go code even contains two paths itself (the main indexer and the `identity_fetcher` agent) that do similar work. This duplication has led to confusion and mismatches (as our earlier research showed, the two lists diverged by 113 missing and 32 extra addresses in one epoch due to logic differences [3] [4]). We should consolidate to **one authoritative whitelist-building mechanism** and reuse it, rather than "implementing everything twice or thrice."

Redundancies and Inconsistencies in the Current Code

1. **Dynamic Stake Threshold vs. Hardcoded 10k:** Our earlier analysis found the Go indexer was using a static 10000 iDNA stake threshold for all identities, whereas the Python code used the **dynamic** `discriminationStakeThreshold` (for Humans) from the API [5] [6]. This caused many Humans with stake between the dynamic threshold and 10000 to be dropped by Go but included by Python. We have since partially fixed this in Go – the `isEligibleSnapshot` function now uses the passed-in threshold for Humans and 10000 for Verified/Newbie [7]. This means Go will now include a Human with, say, 9500 iDNA stake if the dynamic threshold is 9000 (matching Python's behavior). We should verify we consistently use the **actual dynamic threshold** from the epoch data when building the list. (The code currently calls `fetchEpochData` to get the threshold and passes it to `buildEpochWhitelist` [8], which is correct as long as that threshold is indeed up-to-date for the epoch.)

2. **Penalized or Not Approved (Bad Flip) identities:** The Python `whitelist_blueprint` explicitly **excludes any identity that was penalized or not approved** (i.e. had a reported bad flip) by checking each address's `ValidationSummary` (`penalized=false` and `approved=true` required) [9] [10]. Originally, our Go indexer did **not** do this at the initial snapshot stage – it only filtered by state and stake, so addresses with bad flips slipped into the Go-generated list [11] [12]. We addressed this by updating `buildEpochWhitelist` to filter them out: after retrieving identities, we mark each address with `penalized` or `FlipReported` flags and include it in the whitelist only if **neither flag is set** [13] [14]. The Go code uses two helper checks: `getPenaltyStatus` (which calls the official API's `/ValidationSummary` to see if the identity was penalized) [15] [16], and `hasFlipReport` (which uses the node's `dna_identity` RPC to see if the flag `AtLeastOneFlipReported` is in the last validation flags) [17] [18]. This covers the same ground as Python's `approved/not approved` check. **Result:** addresses that flipped badly or failed approval are now excluded on

the Go side as well. This resolves the ~32 "extra" addresses that Go had included erroneously before [4].

3. **Multiple Go Agents doing the same work:** We currently run `agents.RunIdentityFetcher` on startup in parallel with our main indexer logic [19] [8]. The `identity_fetcher` code essentially **reimplements the whitelist building in Go** using the blueprint's method: it finds the Short Session blocks, gathers addresses with transactions, fetches bad authors and validation summaries from the API, and writes out a JSON file of eligible addresses [20] [21]. This is nearly the same logic performed by `buildEpochWhitelist` in `main.go`, which uses the node RPC and then filters [13] [14]. Having both run is unnecessary and potentially confusing. For example, `identity_fetcher` writes `data/whitelist_epoch_<N>.json` (as an array of addresses) independently, while `buildEpochWhitelist` also writes a `whitelist_epoch_<N>.json` (with addresses and merkle root) – we could be double-writing or even overwriting files in different formats. More importantly, if one of these paths had a bug or slight difference, our source of truth becomes ambiguous. **Example:** In the current code, `identity_fetcher` applies the same filters (penalized, approved, stake thresholds) in its loop [21] [22], so it likely produces the same set of addresses as `buildEpochWhitelist` does now – but we run both! This is redundant.

4. **Similar helper functions implemented twice:** There are functions in both `main.go` and `agents/identity_fetcher.go` that do similar things – e.g. fetching the current epoch and threshold (`fetchEpochData` vs `getEpochLast`), getting blocks/transactions, etc. This duplication means if we change one, we might forget to change the other. It's better to have one set of well-tested utilities for these tasks.

Consolidation Plan – "Single Source of Truth" for Whitelist

We should **pick one implementation and stick to it** for generating the whitelist each epoch. Given that we have now corrected the Go indexer's logic to match the Python output, the Go `buildEpochWhitelist` is a good candidate for the single source of truth. It integrates with our database (storing snapshots, merkle roots, etc.), and it's triggered at the right times (on startup and on epoch finalization).

Proposal: Use the `buildEpochWhitelist` path as the only whitelist builder, and **retire the separate `identity_fetcher` agent from the main flow.** Concretely:

- **Do not run `RunIdentityFetcher` automatically on startup**. We can remove or disable the call in `main.go` that launches it in a goroutine [23]. This agent was useful for testing or initial prototyping, but now it's duplicative. If needed, we can keep the code for `RunIdentityFetcher` as a manual tool (for example, a command-line utility to rebuild the whitelist strictly from the official API, which could help debugging), but it shouldn't be running concurrently in production. By consolidating, we avoid writing the whitelist twice and ensure the output format is consistent (our server's `/whitelist` endpoints will serve the DB-backed list that `buildEpochWhitelist` produces).

- **Use one data source as primary:** The local **node's RPC (`dna_epochIdentities`)** is very efficient for grabbing all identities at once, so keep that as the primary method within `buildEpochWhitelist`. It returns a list of `{address, state, stake}` for the epoch in question [24] [25]. We then iterate and apply the filtering as we already do. This covers *all* identities in the consensus state for that epoch (i.e. all who were validated and remained active). We no longer need to manually gather addresses via transactions if this RPC is working.

- **Ensure filtering logic is centralized:** The only place we determine eligibility should be inside `buildEpochWhitelist` (and the helper checks it calls). Right now, `buildEpochWhitelist` uses `isEligibleSnapshot(state, stake, threshold)` for the basic stake/state filter and then checks `!penalized && !flip` before adding to the list [13] [14]. This is good. We should be careful that `isEligibleSnapshot` **uses the dynamic threshold correctly** – it does, for Humans, by design [7]. And for Newbie/Verified it uses 10000 (since the threshold concept doesn't apply to them, a flat 10k is the rule). This mirrors the Python logic exactly [26]. Also, any state outside `{"Newbie","Verified","Human"}` will return false in `isEligibleSnapshot` (and Python similarly excludes "wrong state" identities [27]). So the criteria are uniformly enforced now in one place.

- **Verify no other code path is bypassing this logic:** For example, if we had a SQL query to fetch "eligible" identities directly, we should remove or update it. It appears we previously had a `queryEligibleSnapshots` or similar in older code that **hardcoded 10000** for Humans [28]. That approach should be fully replaced by the new `buildEpochWhitelist` + DB storage of the results. Any endpoints (like `/whitelist` or `/eligibility`) should use the `currentWhitelist` or the stored snapshot, not recalc with outdated logic. (A quick check shows our `/whitelist` handlers use the prepared list/DB, so we're fine, and we didn't find any remaining hardcoded-threshold queries in the current code.)

By consolidating to one builder, we avoid implementing the same filters in multiple places. This makes maintenance easier – if the Idena protocol rules change (say a stake threshold change or new identity state), we'll update the filtering once in `buildEpochWhitelist` and know that applies everywhere.

Fixing Initial Data Population via Official API (Epoch Bootstrap)

We still need to handle the **initial filling of the indexer data**, especially if the service starts up fresh in the middle of an epoch or if the local node is not fully synced. In such cases, relying solely on `dna_epochIdentities` might fail or return incomplete data. This is where the official API can help as a fallback. We have all the necessary cURL calls and logic in the `whitelist_blueprint` (and mirrored in our `identity_fetcher`) to do a one-time pull for the current epoch's whitelist.

Suggested fallback mechanism: If `fetchEpochIdentities(epoch)` fails or returns an unexpectedly low count (indicating the node didn't give us the data), then automatically **use the official API to populate the snapshot** for that epoch. We can integrate this into `buildEpochWhitelist` or in the startup logic:

- Use the official **Idena Indexer API** endpoints to get the required data. Notably, the official API does not yet offer a single "get all identities" endpoint (there's an open issue for `/Epoch/{epoch}/Identities` [29] but not implemented), so we must reconstruct the list by scanning blocks – exactly what our Python script does. The steps are:
- **Get epoch info**: Call `/api/Epoch/Last` to get the latest epoch number and the `discriminationStakeThreshold` [30]. Let `currentEpoch` be the latest epoch; we will build the whitelist for `currentEpoch` (which reflects the results of `lastEpoch = currentEpoch - 1` validation). We should also fetch `/api/Epoch/`

`{lastEpoch}` to get details like `validationFirstBlockHeight` [31]. This gives us the starting block of the last validation ceremony.
- **Find ShortSession start block**: Starting from a little after `validationFirstBlockHeight`, search for a block with the `"ShortSessionStarted"` flag. Our Python code does this by checking blocks in a range (+15 or so) [32]; our Go fetcher did it by scanning backwards from the chain tip [33]. Using the epoch info is more direct: for example, take `firstBlockHeight + 15` as a guess and scan forward ~20 blocks until we see `ShortSessionStarted`. We can use the official API `Block/{height}` endpoint to inspect block flags (the blueprint does this, treating missing flags as empty) [34] [32]. Once we find the block where short session starts, we know validation-related transactions begin from there.
- **Collect addresses from validation TXs**: Iterate over blocks from the short session start through the next several blocks to capture all validation transactions. The blueprint collected 7 blocks of txs by default [35] [36]. In Go, our agent did similarly (it looped until it found 7 blocks with transactions after short session) [37] [38]. Using the official API, we'd call `/api/Block/{height}/Txs` (with pagination if needed) to get all transactions in each block [39] [40]. We gather the unique sender addresses (`tx.from`) from those transactions. These addresses represent all identities that **participated in the validation** (submitted flips or answers) – which is effectively the superset of identities that *could* be eligible for rewards or group membership. (Any identity that did not send a validation transaction either missed the validation – thus will be penalized and not eligible – or was not validated at all.)
- **Fetch bad flip reporters**: Call the official endpoint `/api/Epoch/{lastEpoch}/Authors/Bad` to get the list of addresses that were reported for bad flips in that epoch. Our Python prototype does this and accumulates them into a set [41] [42]. These are the identities with **approved=false** in the validation (they made at least one bad flip). We'll use this to exclude them.
- **For each candidate address, get final validation outcome**: Call `/api/Epoch/{lastEpoch}/Identity/{address}/ValidationSummary` for each address we collected [43]. This gives the identity's final state (`Newbie/Verified/Human/...`), stake, and flags like `approved` and `penalized` [44] [45]. With this data, apply our filters:
    - If `penalized == true` or `approved == false` (meaning they either failed or had a bad flip), **exclude** the address [46] (the Python code prints an exclusion reason for these cases). In our Go terms, these correspond to `penalized` or `FlipReported` flags.
    - Check the identity's **state and stake** against thresholds: If state is `"Human"`, require `stake >= discriminationStakeThreshold` [26]; if state is `"Verified"` or `"Newbie"`, require `stake >= 10000`. If an address's stake is below the cutoff for its state, exclude it. If the state is something else (e.g. `"Suspended"`, `"Zombie"`, etc.), exclude as "wrong state" [27].
    - Any address that passes all these checks is eligible and goes into the whitelist. (This is exactly what `buildEpochWhitelist` does internally when we have the data – we're just reproducing it via API calls.)
- **Insert the results into our system**: We should create the `EpochSnapshot` entries for each fetched identity (including those we exclude, if we want a full record) and then compute the final list and Merkle root as usual [47] [48]. Essentially, we can call the same `upsertEpochSnapshots` and `saveMerkleRoot` functions with the data we obtained, so that the rest of the system (DB and endpoints) believes `buildEpochWhitelist` ran.

By doing the above, we leverage the official API to **bootstrap our indexer's data when the local node isn't ready or reliable**. We already have a lot of this logic in code form (in `agents/identity_fetcher.go`). In fact, `RunIdentityFetcherOnce` performs steps very similar to 3–5 above using the node's API URL (which can be pointed to the official API) [37] [21]. We can reuse those helper functions: e.g. `fetchBadAuthors(nodeURL, apiKey, epoch)` [49] and `fetchValidationSummary(nodeURL, apiKey, epoch, addr)` [50], which are currently designed to hit the `/Authors/Bad` and `/ValidationSummary` endpoints. To avoid circular dependency, we might move them into a common package or make them accessible from `main`. But the heavy lifting (parsing JSON, handling continuation tokens) is already written and tested in those functions.

When to trigger this fallback: The simplest trigger is if `fetchEpochIdentities` returns an error or empty result. For example, in `buildEpochWhitelist`, if `ids` comes back nil/error [51], we catch that and then invoke an "API fetch" routine. Alternatively, we could proactively use the API if the local node's reported current epoch is different from what the official API says (signaling the node is behind). However, simply catching the error is likely enough. On a fresh start, if the node isn't up or synced, `http.Post` to `dna_epochIdentities` will fail – at that point we log a warning and can call a `fallbackBuildEpochWhitelistViaAPI(epoch)` function implementing the above steps.

One thing to note is that using the official API requires no special API key (it's public), but if our config has `IDENA_RPC_KEY`, we append it to the requests for local node. We should ensure not to append an empty `apikey` param to official calls (the functions already handle that by checking if `apiKey` is non-empty [49] [52]).

With this in place, the **initial indexer fill will be robust**. For example, if our service starts at epoch N and our node is still catching up, we'll fetch the epoch N whitelist via official API, seed our database, and serve it immediately. Once the local node syncs, future incremental updates (like real-time new flips, etc., which in our case mostly happen at epoch boundaries) will still be handled by `watchEpochFinalization`.

Summary of Changes to Implement

1. **Eliminate duplicate whitelist builders:** Use only `buildEpochWhitelist` for generating the whitelist each epoch. Remove the automatic running of `identity_fetcher`. Also, any one-off tools (such as a `strictbuilder` command, if it exists) should pull from the same logic or be removed to avoid confusion. This ensures we're not "implementing the same thing three times." The code already in `buildEpochWhitelist` covers all the needed filtering, especially after our recent fixes (dynamic threshold for Humans, penalized/flip exclusions). Consolidation will prevent divergence.

2. **Integrate Official API Fallback for epoch bootstrap:** Enhance `buildEpochWhitelist` (or the startup process) to detect when the local node cannot provide the data, and then use the official API to fetch the identities and their validation outcomes. We have the blueprint's cURL recipes and even in-code helpers to do this. Essentially, incorporate the logic from `RunIdentityFetcherOnce` [37] [21] directly into our indexer when needed. This may involve a bit of refactoring (making sure we can call those functions from `main.go`), but it's straightforward.

3. **Consistency in data and output:** After these changes, the **whitelist should be identical whether built via local RPC or via official API**. It's worth double-checking by comparing outputs with the Python script for a given epoch (as we did before). With our fixes, we expect the lists to match exactly. The earlier 113-address and 32-address discrepancies [53] [54] should be resolved. All Humans above the *dynamic* threshold will be included, and no penalized/not-approved identity will slip in. This one unified list (stored in DB and exposed by our `/whitelist` endpoints) will be the single source of truth.

4. **Remove or adjust redundant storage/output:** Since we'll trust the DB and in-memory `currentWhitelist`, we don't necessarily need to write a separate JSON file in the `identity_fetcher` format. We might stop writing the `data/whitelist_epoch_N.json` in that agent (or remove the file output there) to avoid confusion. Our `buildEpochWhitelist` already writes a JSON with addresses and merkle root [55] for persistence. Perhaps we keep just that. The key is that there is one definitive JSON per epoch, not two slightly different ones.

By **consolidating and simplifying**, we reduce maintenance overhead and the risk of one part of the system diverging from the intended logic. Going forward, any rule change (say, if the stake requirement or validation flags change) will be implemented once in `buildEpochWhitelist`. This will make the system more reliable.

Finally, after making these changes, we should thoroughly test on a new validation cycle to ensure the initial fill works (try starting the service from scratch just after an epoch finalization – it should fetch from official API and populate the whitelist correctly). We should also verify that during normal operation, when the local node is synced, the behavior is unchanged (the whitelist builds on finalization and matches expectations). The result will be a clean, single implementation of the whitelist generation that aligns with the official definition and our Python blueprint reference.

Sources:

- Comparison of Python vs Go whitelist logic (dynamic threshold & flip/penalty filtering) [5] [6]
- Go implementation of whitelist filtering (`isEligibleSnapshot` and final checks) [13] [14]
- Duplicate filtering logic in the Go fetcher agent (to be consolidated) [21] [22]
- Python blueprint outline of steps using official API (fetch blocks, threshold, filter flips, stakes) [1] [2]
- Python code fetching bad authors and checking validation summaries (official API usage) [41] [44]
- Note on missing official endpoint for listing identities (GitHub issue for `/Epoch/{epoch}/Identities`) [29]

---

[1] [2] README.md
https://github.com/ubiubi18/whitelist_blueprint/blob/793441a12cf9bc89b52455f8605755b129a06313/README.md

[3] [4] [5] [6] [9] [10] [11] [12] [28] [53] [54] Python vs Go Whitelist Logic.pdf
file://file-A1jAXU8zVZaGuNY2u7PZ6h

[7] [8] [13] [14] [15] [16] [17] [18] [19] [23] [24] [25] [47] [48] [51] [55] main.go
https://github.com/ubiubi18/IdenaAuthGo/blob/b8ab01a62ede39cb238ca4c5241103c807887fe7/main.go

[20] [21] [22] [33] [37] [38] [49] [50] [52] identity_fetcher.go
https://github.com/ubiubi18/IdenaAuthGo/blob/b8ab01a62ede39cb238ca4c5241103c807887fe7/agents/identity_fetcher.go

[26] [27] [30] [31] [32] [34] [35] [36] [39] [40] [41] [42] [43] [44] [45] [46] build_idena_identities_strict.py
https://github.com/ubiubi18/whitelist_blueprint/blob/793441a12cf9bc89b52455f8605755b129a06313/build_idena_identities_strict.py

[29] Issues · idena-network/idena-indexer-api - GitHub
https://github.com/idena-network/idena-indexer-api/issues