Mosaic Code Review: Executive Summary

Introduction

The Mosaic team engaged BlockDeep Labs to conduct a thorough review of selected parts of their Substrate-based code. This review aimed to assess the efficiency and maintainability of the codebase while identifying areas for improvement. BlockDeep Labs has analyzed critical components, including the on_idle hook, session-ending logic, data structures, and overall code quality. The following executive summary presents key findings and actionable recommendations to enhance Mosaic's implementation.

Findings

1. Data Structures and Efficiency

- Session endings are processed in a first-in-first-out (FIFO) fashion. Using a standard array has O(n) complexity for head removals. Replace it with a queue to achieve O(1) insertions and deletions.
- Replace raw mathematical operations with checked or saturated arithmetic to prevent unexpected overflows or underflows.

2. Weight Management and Ordering

- If remaining weight is greater than zero but still insufficient for the *on_idle* operation, return early to avoid block exhaustion.
- Since *MaximumBoundValidators* is not strictly enforced, relying on it for weight calculations is inaccurate. Instead, use a *CountedStorageNMap* or similar structure to record the exact number of validators.
- Consider performing slashing before delivering rewards, as slashing has higher priority for chain stability.

3. API and Code Clarity

- In *reward_contract*, accept *total_v_imbalance* as a mutable reference (rather than returning a balance) for consistency and clarity.
- Remove the NFT count parameter from the benchmark, since reward logic does not depend on the number of NFTs a contract holds. Always benchmark with *MAX_NFTS_PER_CONTRACT*.

4. Error Handling and Logging

- Replace unwrap and expect statements with robust error handling; unexpected missing values could stall the chain.
- Use error logs (with additional context) instead of debug logs when critical issues arise to aid production troubleshooting.

5. Testing and Coverage

- Add tests covering stalled session endings to ensure the process can gracefully resume if delayed.
- Increase coverage for lines where values are read but not used, as uncovered logic can hide potential bugs.

6. Alternative Approach: Using *on_initialize*

- There is no guarantee that *on_idle* will run with a certain frequency or have enough weight for all operations. As an alternative, place the session-ending logic in *on_initialize*, compute remaining block weight, and potentially "lock" new extrinsics until session-ending tasks finish. This increases predictability and stability at the cost of short-term extrinsic restrictions.

7. Documentation and Terminology

- Clearly explain the purpose and rationale of using *on_idle* (instead of *on_initialize*), including how it handles remaining block weight and why it was chosen.
- Add explanatory comments for multiple components (e.g., *SessionEnding*, *Context*, *SweepContext*) to help maintainers and external developers understand their intended usage.
- Avoid using the term "hook" for features that are not standard Substrate pallet hooks. Use "handler" or similar wording to reduce confusion.

Conclusion

The Mosaic codebase demonstrates solid functionality but requires improvements in documentation, error handling, and efficiency. Optimizing data structures, refining block weight management, and ensuring robust logging will enhance the system's resilience and performance. Additionally, adopting *on_initialize* instead of *on_idle* may provide greater execution predictability. Implementing these recommendations will help Mosaic achieve better scalability, maintainability, and long-term reliability for their chain.

Approval

I, the undersigned, approve the findings and recommendations outlined in this report.

Name: **Gautam Dhameja**

Title: **Managing Director, BlockDeep Labs**

Signature: 

Date: **06 Feb 2025**

Substrate Code Review Feedback

Prepared By: BlockDeep Labs
Prepared For: Mosaic Protocol

Team: Mosaic
Review Date: 18-11-2024
Report Date: 26-11-2024

Commit Hash: ea2c5741d333bd6e55dc67a25814ff195c64edab
Review Scope Items:

Reviewer Feedback:

Assessment:

Documentation and Codebase Quality:	**Feedback:**
a. Is there sufficient documentation for compiling and executing the code?	Yes, documentation is sufficient.
b. Does the project documentation provide a conceptual understanding of the project?	Yes, the project is fully documented.
c. Are there written tests for the codebase, and do they run successfully, indicating completed code components?	There exist written tests, they run successfully and the coverage is sufficient for the logic. However, benchmarks are not tested, and they do not include post-execution checks.

Code Review Comments:

NFT Delegation Pallet

Generic

1. Crate-level documentation starts with "//!".
2. The pallet is tightly coupled with FRAME's NFT pallet. However, there exist several trait implementations that allow decoupling. We highly recommend always refraining from using tight coupling when possible and instead adding another configuration item that implements any relevant trait needed in the pallet.
3. The *NftExpirationHandler* configuration item is not properly documented. It would help the reader to know what this handler is for and which part of the system is meant to use it.
4. Storage getters are deprecated and should be removed from all storage items.
5. There are several problems with *PalletAccountId*:
    a. It is only set on genesis. However, the pallet might not be included in chains at genesis, leaving that item unset. **This is a major issue.**
    b. There could be a runtime upgrade that changes the *PalletId* configuration item, while the storage item would remain untouched unless specifically migrated. **This is a major issue.**
    c. In general, it is a redundant storage item, as its value can be directly derived from the *PalletId* configuration item.

   d. Lastly, accessing its value has a penalty of a storage read, while it is much faster to obtain its value via *T::PalletId::get().into_account_truncating()*.

   For the aforementioned reasons we highly recommend removing that storage item and replacing it with the corresponding *PalletId* configuration item. There could be a specific getter function for that matter that makes its access easier both internally and externally. Furthermore, in order to query the pallet's pot account, a runtime API function could be exposed.

   Alternatively, if the pallet's pot account is expected to change, the configuration parameter could be removed and an additional permissioned extrinsic should be added to change its value.

6. There are several structs without proper documentation. Documentation helps readers and developers understand what is the purpose of the components and how they fit in the system.
7. Errors are not documented. Lack of documentation on error variants can make it challenging for developers to understand and address issues. Adding documentation for each error provides valuable context in code metadata, making it easier to identify the cause of failures.
8. *CollectionId* has several flaws:
   a. It might cause confusion, as there is a similar type in the NFTs pallet.
   b. It is only set at genesis, which implies it can only be added to an existing chain by performing a migration. **This is a major issue.**

   We recommend creating a permissioned extrinsic to dynamically set this storage value.
   If this collection is not expected to change, it could be created in advance and added as a configuration parameter instead.
9. *CurrentSession* could and should be a constant to be able to be queried.
10. When creating the collection at genesis the pallet's pot account must be funded with sufficient balance to create the collection. Such a requirement is not specified in the documentation nor is intuitive.
11. No event gets triggered at genesis, especially when minting items. Consider using *do_mint_delegator_token* instead of rewriting the whole minting logic.
12. There are a few types that could have an alias to easily spot them:
    a. *<T as NftsConfig>::ItemId -> ItemIdOf<T>*
    b. *<T as NftsConfig>::CollectionId -> CollectionIdOf<T>*
13. This pallet assumes NFT item IDs are numeric, and hence the *Incrementable* constraint. However, that is not necessarily the case and this can produce unsatisfied constraint errors when integrating the pallet. In any case this constraint should be documented so that teams integrating it are aware of this limitation.
14. Benchmarks are not tested, nor post-execution checks are added. Make sure to:
    a. Add the *impl_benchmark_test_suite*! macro at the bottom of the benchmarks file.
    b. After executing an extrinsic inside a benchmark, add relevant post-execution checks that make sure the state has been altered properly.

c. Add all required pallets to the mocked runtime for benchmark tests to compile and execute.

Extrinsics

mint_delegator_token

1. There is a naming inconsistency in the *nominal_value* lambda's parameter, as it should be renamed to *nominal_value_bytes* to align with other occurrences.
2. This pallet emits an event when a new token is created. However, the NFT pallet itself already emits an event in the same circumstances. We recommend reviewing whether the event emitted by this pallet is redundant or not.
3. The expiration should be checked to be strictly greater than zero, as otherwise it would be de-facto useless, since in order for the owner account to be eligible as a collator it would have to wait until the current session to end, and by that time their token would immediately expire.
   Furthermore, we recommend adding a minimum expiration parameter to the runtime configuration so that no tokens with an expiration lower than a given value can ever be minted.

NftDelegation implementation

1. It would help a lot to document this implementation so that readers can easily understand its purpose.
2. Similarly, all *NftDelegation* methods should include the relevant documentation.
3. If *MaxExpirationsPerSession* is zero, this line will panic, causing the chain to stall. Although it is very unlikely that the runtime will ever be configured as such, it is an unnecessary risk that could be very easily alleviated by simply raising a *TooManyExpirationsPerSession* error. **This is a major issue.**
   Furthermore, we recommend adding a *try_state* hook that checks *MaxExpirationsPerSession* is strictly greater than zero.
4. The *encode_status* function name is not appropriate, as it seems to be performing an encoding operation, when it is actually also storing the result. A more appropriate name would be *set_status*.
   The same actually happens in cases like *decode_nominal_value*. We recommend using the get/set or retrieve/store naming conventions to indicate there are storage reads or writes being performed.
5. Consider adding the expiration session to the *TokenBound* event for added clarity.
6. When binding or unbinding the nominal value is returned, but that NFT attribute itself is not related to the binding or unbinding operations and could be manually retrieved using the *nominal_value* method.

SessionHook implementation

1. The *SessionHook* trait has a fundamental error: it is essentially a wrapper around the *SessionManager* of *pallet-session*, and the *SessionManager* is always called during *pallet-session*'s on_initialize hook. You can check how here the session is rotated in the *on_initialize* hook, which will internally call the session handler right before finishing the session, when the session ends, and

when the next session starts.
We can see how the *validator-subset-selection* pallet implements the SessionManager trait and **unwraps the error** at:
   a. The genesis session.
   b. When a session ends.
   c. When a new session starts.
   d. When a new session is being created.

This is a critical issue, as the hook is prepared to raise errors when it is actually being used in a context where errors cannot occur. To fix this, disallow any kind of errors in the hook, or gracefully handle errors inside the validator-subset-selection pallet.

2. Since this is run inside a hook, there is no storage rollback in case of failure (regardless of the fact that a failure would provoke the chain to stall). Hence, during the expired tokens loop, if any of them would fail, the rest of the tokens for that session would not be processed, leading to a situation where some of them would be processed and their storage properly updated, and others that would not.
To alleviate this, gracefully handle any error that may be produced during the loop and make sure all items are processed.

NFT Permission Pallet

Generic

1. The pallet is tightly coupled with FRAME's NFT pallet. However, there exist several trait implementations that allow decoupling. We highly recommend to always avoid tight coupling when possible and instead add another configuration item that implements any relevant trait needed in the pallet.
2. Storage getters are deprecated and should be removed from all storage items.
3. There are several structs without proper documentation. Documentation helps readers and developers understand what is the purpose of the components and how they fit in the system.
4. Errors are not documented. Lack of documentation on error variants can make it challenging for developers to understand and address issues. Adding documentation for each error provides valuable context in code metadata, making it easier to identify the cause of failures.
5. *CollectionId* has several flaws:
   a. It might cause confusion, as there is a similar type in the NFTs pallet.
   b. It is only set at genesis, which implies it can only be added to an existing chain by performing a migration. **This is a major issue.**
6. When creating the collection at genesis the pallet's pot account must be funded with sufficient balance. Such a requirement is not specified in the documentation nor is intuitive.
7. There are several problems with *PalletAccountId*:
   a. It is only set on genesis. However, the pallet might not be included in chains at genesis, leaving that item unset. **This is a major issue.**
   b. There could be a runtime upgrade that changes the *PalletId* configuration item, while the storage item would remain untouched

unless specifically migrated. **This is a major issue.**

c. In general, it is a redundant storage item, as its value can be directly derived from the *PalletId* configuration item.
d. Lastly, accessing its value has a penalty of a storage read, while it is much faster to obtain its value via *T::PalletId::get().into_account_truncating()*.

For the aforementioned reasons we highly recommend removing that storage item and replacing it with the corresponding *PalletId* configuration item. There could be a specific getter function for that matter that makes its access easier both internally and externally. Furthermore, in order to query the pallet's pot account, a runtime API function could be exposed.

Alternatively, if the pallet's pot account is expected to change, the configuration parameter could be removed and an additional permissioned extrinsic should be added to change its value.

8. This pallet assumes NFT item IDs are numeric, and hence the *Incrementable* constraint. However, that is not necessarily the case and this can produce unsatisfied constraint errors when integrating the pallet. In any case this constraint should be documented so that teams integrating it are aware of this limitation.
9. Benchmarks are not tested, nor post-execution checks are added. Make sure to:
    a. Add the *impl_benchmark_test_suite*! macro at the bottom of the benchmarks file.
    b. After executing an extrinsic inside a benchmark, add relevant post-execution checks that make sure the state has been altered properly.
    c. Add all required pallets to the mocked runtime for benchmark tests to compile and execute.
10. There are a few types that could have an alias to easily spot them:
    a. *<T as NftsConfig>::ItemId -> ItemIdOf<T>*
    b. *<T as NftsConfig>::CollectionId -> CollectionIdOf<T>*

Extrinsics

mint_permission_token

1. This pallet emits an event when a new token is created. However, the NFT pallet itself already emits an event in the same circumstances. We recommend reviewing whether the event emitted by this pallet is redundant or not.

NFTStaking implementation

1. Missing triple slash in the docs.
2. There should be documentation either at the trait definition or at this particular implementation for all functions to better understand their purpose.
3. These lines could be rewritten as follows for better readability:

```
BoundTokens::<T>::mutate_exists(account_id, f: |maybe_item : &mut Option<<...>::ItemId>| {
    if maybe_item.is_some() {
        return Err(Error::<T>::AlreadyBound.into());
    }
    *maybe_item = Some(*item_id);
```

4. There are several functions that should be renamed to better reflect the fact that they are accessing the storage. For example:
    a. *encode_issued_nominal_value -> store_issued_nominal_value*
    b. *encode_permission -> store_permission*
    c. *encode_nominal_value -> store_nominal_value*
    d. *decode_issued_nominal_value -> retrieve_issued_nominal_value*
    e. *decode_nominal_value -> retrieve_nominal_value*
    f. *decode_permission -> retrieve_permission*
5. These lines could be rewritten for better readability:

```
fn unbind(account_id: &T::AccountId) -> Result<T::Balance, DispatchError> {
    BoundTokens::<T>::mutate_exists(
        account_id,
        f: |maybe_item : &mut Option<<...>::ItemId>| -> Result<T::Balance, DispatchError> {
            let item_id : <T as Config>::ItemId = maybe_item.ok_or(Error::<T>::NotBound)?;
            let collection_id : <T as Config>::CollectionId =
                Self::collection_id().ok_or(Error::<T>::CollectionNotInitialized)?;

            *maybe_item = None;
            <NftsPallet<T> as Transfer<_>>::enable_transfer(&collection_id, &item_id)?;

            Self::deposit_event( event: Event::<T>::TokenUnbound { item_id });
            Self::decode_nominal_value(&collection_id, &item_id)
        },
    )
}
```

NFT Staking Pallet

Generic

1. The pallet is apparently reserving the native asset, while the naming "lock" is used. Locking and reserving are two different actions that yield different outcomes. To be more specific:
    a. Reserve/hold: Reserved balances are set aside and cannot be spent, but they still count toward the account's total balance. This ensures funds are allocated for specific purposes, such as deposits, without removing ownership.
    b. Lock/freeze: Locked funds remain in the account and visible, but they cannot be spent or transferred until the lock expires or is removed. Unlike reserve, the funds are not explicitly earmarked.

    Review if reserving is the intended operation here, as traditionally staking is associated with locking and not reserving.
2. Pallet configuration items (types, constants, or associated items within the Config trait) that lack documentation can confuse developers and users. Without clear descriptions, it becomes harder to understand their purpose, leading to misconfigurations or inefficient implementations.
3. Events emitted by the runtime lack proper documentation, making it harder for users to understand their purpose. Additionally Event documentation for each error provides valuable context inside the code metadata allowing frontend clients and users to get more insight regarding an event that is emitted.
4. Lack of documentation on error variants can make it challenging for developers to understand and address issues. Adding documentation for each error provides valuable context in code metadata, making it easier to identify the cause of failures.
5. The *MAX_NFTS_PER_CONTRACT* constant has an arbitrary value that might be subject to change in the future. We recommend adding this value as another constant runtime configuration parameter to the pallet for added flexibility and maintainability.
6. Storage items are not properly documented, which makes it difficult for developers to understand how data is structured, accessed, or manipulated. This can lead to misusing storage entries, causing bugs or inefficiencies in runtime execution. Additionally, it creates barriers for collaboration, as contributors must reverse-engineer functionality instead of focusing on building new features.
7. In the genesis config the *NFTStakingHandler* is used several times. It is presumably configured to be the NFT Permission pallet, and that implies that it must be initialized before the NFT staking pallet, as otherwise the chain won't be able to produce its genesis block. For that, the NFT Permission pallet must be defined in the runtime before the NFT Staking pallet, and such a limitation should very likely be documented.

8. If there are repeated validators at genesis, such an error should be properly described in the *assert* macro.
9. There is a typo here.
10. There are several storage mappings whose key is the validator's account ID. Consider if some or all of those maps could be encapsulated in a single struct so that the pallet could save some storage reads and writes.
11. The pallet seems to allow not to have initial staking validators at genesis. Check that does not impact block production by yielding an empty collator set.
12. We recommend using the *is_zero* method from the *Zero* trait instead of comparing against zero, as done here.
13. Benchmarks are not tested, nor post-execution checks are added. Make sure to:
    a. Add the *impl_benchmark_test_suite*! macro at the bottom of the benchmarks file.
    b. After executing an extrinsic inside a benchmark, add relevant post-execution checks that make sure the state has been altered properly.
    c. Add all required pallets to the mocked runtime for benchmark tests to compile and execute.
14. The *is_chilled* function panics in case the validator's account ID is not included in the map. This behavior must be changed, as anyone could call, for example, the *unbind_validator* extrinsic and the chain would stall. **This is a major issue.**
15. Something similar occurs in the *is_slacking* function. **This is a major issue.**
16. The *unlock_currency* function also unwraps a possible error. **This is a major issue.**
17. Extrinsics are not documented. This can lead to misunderstandings about their functionality, expected inputs, permissions, and possible errors. Proper documentation ensures that developers and users understand how to interact with the extrinsic correctly. Additionally, it provides valuable context inside the code metadata allowing frontend clients and users to get more insight regarding an event that is emitted.
18. We recommend the usage of saturating or checked arithmetical operations instead of the raw ones. Please review these cases:
    a. Subtraction 1
    b. Addition 1
    c. Addition 2

Extrinsics

bind_validator

1. The *Validators* storage item is firstly read and later written. This process can be improved by using *try_mutate* or *try_mutate_exists*.
2. This check could be placed earlier to avoid performing operations that will be rollbacked in case of error.

3. Although unlikely to happen, this saturated addition will not properly count the accumulated stake if such value produces an overflow. To alleviate this, *checked_add* could be used.
4. Both grow_total_validator_stake_by and request_new_contract are modifying the exact same storage item, and hence reading and writing from/to it twice. This operation could be improved by bundling both operations in a single one to improve performance.

unbind_validator

1. The code assumes there must always exist a one-to-one correspondence between accounts and validator IDs. However, that is an assumption entirely dependent on how *pallet-session* has been configured, and hence this unwrapping operation is not necessarily true in all cases. Failing to find a correspondence between these types implies the chain will stall. **This is a major issue.**
   We recommend gracefully handling the error instead of unwrapping.
2. The operation performed here is unbounded and with algorithmic complexity equal to O(n), as the session pallet maintains unbounded lists both for the queued keys and the validator set. Unbounded iterations over large data structures can lead to resource exhaustion and may result in denial of service (DoS) attacks if the process consumes too many resources.
   In this case, there exists the possibility that validators cannot unbind, as an unbounded operation could potentially consume all block weight, provoking the extrinsic to fail out of resource exhaustion. Still, we acknowledge that this is a rare scenario, since there are a set of custom pallets that we assume will make sure the validator set is reasonably sized.
3. Instead of performing a conversion from *u32* to *usize*, it would be better to do the opposite operation, as shown below.

```
ensure!(
    contracts.len() as u32 <= contract_count,
    Error::<T>::MoreContractsExist
);
```

   This way it avoids unwrapping a value and hence stalling the chain in case of failure.
4. It is not intuitive what the *contract_count* parameter is for. Adding extrinsic documentation would greatly help the reader understand what is the purpose of this check.

5. The weight for this function is not necessarily correctly calculated, as it depends on the user-provided *contract_count* parameter, which can be strictly higher than the actual number of contracts for the caller. To determine the exact consumed weight this extrinsic should return a *DispatchResultWithPostInfo*. See the following example:

```
    let contract_len = contracts.len() as u32;
    ensure!(
        contract_len <= contract_count,
        Error::<T>::MoreContractsExist
    );

    // Note: this includes the self-contract as well
    for (contractee, contract) in contracts {...}

    // Note: we can remove immediately as no reward would be given.
    // Caveat: slashes in current session? We can forgive them, right?
    Validators::<T>::remove(&caller);
    ValidatorStates::<T>::remove(&caller);
    TotalValidatorStakes::<T>::remove(&caller);
    InverseSlashes::<T>::remove(&caller);

    Self::deposit_event(Event::<T>::ValidatorUnbound(caller));

    Ok(Some(<T as Config>::WeightInfo::unbind_validator(contract_len)).into())
}
```

enable_delegations

1. If using mutate, the value might be mutated prior to returning an error, and that change will be persisted in storage. To avoid that, use *try_mutate* instead, since it will only mutate the value if *Ok* is returned.

disable_delegations

1. If using mutate, the value might be mutated prior to returning an error, and that change will be persisted in storage. To avoid that, use *try_mutate* instead, since it will only mutate the value if *Ok* is returned.

chill_validator

1. If using mutate, the value might be mutated prior to returning an error, and that change will be persisted in storage. To avoid that, use *try_mutate* instead, since it will only mutate the value if *Ok* is returned.
2. We recommend emitting events after the storage has been properly altered to reflect the change, and never before. This is not the case of the chill_state function when used in this extrinsic, and we recommend refactoring it so that

it does not emit any event, and instead it gets emitted right after the storage is altered.

unchill_validator

1. If using mutate, the value might be mutated prior to returning an error, and that change will be persisted in storage. To avoid that, use *try_mutate* instead, since it will only mutate the value if *Ok* is returned.
2. This check should likely be placed before mutating the state to avoid performing operations that will be rollbacked on failure.

self_stake_currency

1. The *do_stake_currency* function should be the one emitting the *CurrencyStaked* event as the last part of its execution for it to be self-sufficient.
2. This saturated addition may yield an inconsistent state in case of overflow. To alleviate this, we recommend using *checked_add* and returning the corresponding error in case of failure.

self_stake_nft

1. The NftDelegated event is not aligned with the extrinsic name. We suggest renaming the extrinsic to *self_delegate_nft* and the *do_stake_nft* function to *do_delegate_nft*, as NFTs are not actually "staked", but delegated.
2. The *do_stake_nft* function should be one emitting the *NftDelegated* event as the last part of its execution for it to be self-sufficient.
3. In case the delegated_nfts attribute changes to a configurable parameter in the pallet this force_push would yield an invalid state. To alleviate this, make sure you use *try_push* and return an error in case of failure.

self_unstake_currency

1. The *do_unstake_currency* function should be the one emitting the *CurrencyUnstaked* event as the last part of its execution for it to be self-sufficient.
2. Make sure you use *try_mutate* here, as otherwise the storage could be modified regardless of the lambda's outcome.
3. We recommend using *saturating_reduce* to improve readability here.
4. Increasing the unlocking value may yield an inconsistent state if there is an overflow. To alleviate this, use *checked_add* in conjunction with *try_mutate_exists*.

self_untake_nft

1. The *do_unstake_nft* function should be the one emitting the *NftUndelegated* event as the last part of its execution for it to be self-sufficient.

2. The NftUndelegated event is not aligned with the extrinsic name. We suggest renaming the extrinsic to *self_undelegate_nft* and the *do_unstake_nft* function to *do_undelegate_nft*, as NFTs are not actually "unstaked", but undelegated.
3. Performing a linear search over a bounded array is an acceptable operation, but that can be improved by using a *BoundedBTreeMap* if items have to be queried by a unique identifier.
   Alternatively, items could be sorted inside the array. This way we can perform a binary search.

set_minium_staking_period

1. There is a typo in the extrinsic's actual name.
2. Although the variable is mutated at the end of the lambda function, we recommend always using *try_mutate* to avoid unintended invalid state transitions.

set_commision

1. Although the variable is mutated at the end of the lambda function, we recommend always using *try_mutate* to avoid unintended invalid state transitions.

kick

1. This extrinsic definitely requires extensive documentation, as it is potentially a destructive operation and to comprehend the reasons and limitations it may have is crucial.
2. Although the variable is mutated at the end of the lambda function, we recommend always using *try_mutate* to avoid unintended invalid state transitions.

topup

1. There is no need to mutate the contract here, since it is only read.

Implementations

1. There is a double loop in the validators function, which implies a potentially unbounded operation that could consume more weight than available in a single block. For now there seems not to exist a straightforward solution, as validator IDs are not sortable. We recommend keeping boundaries sufficiently low for this operation to be kept under control.
2. We recommend renaming this parameter to *maybe_validator*, since it is an optional value.
3. This operation may cause the chain to stall. We recommend discarding the reward if an error occurs here by assigning a reward of zero. Something similar occurs here, and the same solution is proposed.

4. We want to raise awareness about how rewards are distributed. The original idea is that for every validator all its stakers are rewarded when the session ends. While the number of validators per session might be under control, there are no safety guarantees regarding the number of stakers per validator, since it is an unbounded double map. For that reason **the system clearly does not scale and the chain may run out of weight when switching sessions.** Since this operation is performed in the session pallet's *on_initialize* hook, that implies the **chain might not be able to produce the block and will stall.** Possible workarounds would be:
    a. Implement an on-demand reward system, where per-session rewards can be calculated and any user can manually retrieve them.
    b. Limit the maximum number of stakers per validator to work with bounded data structures.

Grading:

Scope Item 1: NFT Delegation Pallet

Topic	Grade	Feedback
Documentation	4	The documentation is overall good. Minor improvements have been suggested.
Code Structure	5	The structure is mostly correct.
Code Performance	4	Code performance has a few improvements, especially in terms of storage access.
Tests	5	Tests are correct and have a high coverage.

Scope Item 2: NFT Permission Pallet

Topic	Grade	Feedback
Documentation	4	Documentation is mostly sufficient. There are minor improvements suggested in the comments.
Code Structure	5	Code structure is essentially correct.
Code Performance	4	Minor adjustments could be made to improve storage reads and writes.
Tests	5	Tests are sufficient and provide a decent coverage.

Scope Item 3: NFT Staking Pallet

Topic	Grade	Feedback
Documentation	3	There are several items in the pallet that should be documented.
Code Structure	4	The code structure is overall correct. Minor improvements have been suggested.
Code Performance	4	Code performance is overall sufficient. Some improvements have been suggested that can optimize it.
Tests	5	There is extensive test coverage for this complex pallet.

Supporting Reference for Assessment & Grading:

Criteria:

a) **Deliverable Functionality:** Assess whether the delivered code fulfills its intended purpose, a crucial aspect of evaluation.
b) **Documentation Quality:** Evaluate the comprehensiveness and clarity of documentation accompanying the delivery, aiming for accessibility to the broader community.
c) **Code Structure and Readability:** Examine the layout and clarity of the codebase to ensure it is understandable for community members who may review it.
d) **Performance Optimization:** Investigate the efficiency and performance of the chain, emphasizing avoidance of redundancies and straightforward implementation without the need for extensive benchmarking.
e) **Testing Coverage:** Emphasize the importance of thorough testing by evaluating the extent to which the project incorporates comprehensive testing protocols.

Grading: Scoring 1-5.

Documentation:

- **1) Inadequate:** Absence of documentation.
- **2) Deficient:** Documentation is present but lacks essential components.
- **3) Fair:** Existing documentation poses challenges in comprehension.
- **4) Adequate:** Documentation is comprehensive yet demands moderate effort to grasp.
- **5) Exemplary:** Documentation is impeccably structured, comprehensive, and easily comprehensible.

Code Structure:

- **1) Inadequate:** Code structure is convoluted and difficult to follow.
- **2) Deficient:** Some attempt at segmentation based on objectives, but lacks overall coherence.
- **3) Satisfactory:** Broad segmentation exists, however, with deficiencies in detailed structuring.
- **4) Proficient:** Segmentation of processes and tasks demonstrates good practice, albeit with minor shortcomings.
- **5) Exceptional:** Code division and structure are flawless, facilitating ease of comprehension and replication.

Code Performance:

- **1) Deficient:** Code performance is hindered by extensive repetitive patterns and lack of segmentation.

- **2) Subpar:** Notable redundancy exists, particularly within complex sections of the codebase.
- **3) Acceptable:** Segmentation is apparent, with discernible efforts to minimize redundancy, albeit not fully achieved.
- **4) Competent:** Overall execution is commendable, with minor areas identified for optimization.
- **5) Outstanding:** Code execution demonstrates exceptional performance, characterized by minimal redundancy and optimal efficiency.

Tests:

- **1) Deficient:** Testing procedures are entirely absent.
- **2) Limited:** Minimal testing efforts are evident, often simplistic in nature.
- **3) Adequate:** Testing coverage is satisfactory, though certain scenarios remain untested.
- **4) Proficient:** Testing protocols cover a majority of components adequately, with some areas identified for enhancement.
- **5) Exemplary:** Comprehensive testing which encompasses all conceivable scenarios, reflecting exemplary adherence to testing standards.

Substrate Code Review Feedback

Prepared By: BlockDeep Labs
Prepared For: Mosaic

Team: Mosaic
Review Date: 25-01-2025
Report Date: 29-01-2025

Commit Hash: d67e853ce958286de25855a69c2fa79f1d7a6591
Review Scope Items:

Assessment:

Documentation and Codebase Quality:	**Feedback:**
a. Is there sufficient documentation for compiling and executing the code?	The documentation to assist compilation is sufficient.
b. Does the project documentation provide a conceptual understanding of the project?	Yes, the project includes basic documentation, providing an overview but it's not exhaustive. There is room for improvement.
c. Are there written tests for the codebase, and do they run successfully, indicating completed code components?	There are written tests and they run successfully, providing a decent test coverage.

Code Review Comments:

NFT Staking Pallet

1. It would greatly help the reader to document what the *on_idle* hook is used for, and why it has been chosen instead of *on_initialize*.
2. There are several pieces of code that should be documented to help the reader understand their purposes. Here is a non-exhaustive list of items that we recommend further documenting:
    - *SessionEnding*
    - *Context*
    - *State*
    - *SweepContext*
    - *Sweep*
    - *ValidatorContext*
    - *RewardValidator*
3. Session endings are being accumulated and processed in the same order they occurred whenever there is available weight in the block, which implies they are retrieved from and inserted into the first position. However, a standard array is **not** the most optimal data structure for such operation, since removing the first position has a logarithmic complexity of O(n). In a *no_std* environment the best data structure at our disposal for this matter is the queue, which

guarantees O(1) insertions and deletions in the head and tail. This is not something really severe, given that in a normal scenario the previous session will be fully processed by the time the current session ends, but since that cannot be 100% guaranteed given the nature of the *on_idle* hook, it is better to be as efficient as possible in all scenarios.

4. The term "hook" is usually associated with pallet hooks in the Substrate context. This can turn out to be misleading for developers, so we recommend using the term "handler instead".
5. There exists the possibility that the remaining weight is greater than zero, but less than the cost of the *on_idle* operation itself (which simply measures the overhead). In that case the code would perform a mutate operation that may exhaust the block, although that scenario is highly unlikely to occur. In any case, it is better to return early by checking if the limit is zero.

```rust
#[pallet::hooks]
impl<T: Config> Hooks<BlockNumberFor<T>> for Pallet<T> {
    fn on_idle(_block_number: BlockNumberFor<T>, remaining_weight: Weight) -> Weight {
        let mut meter : WeightMeter = WeightMeter::with_limit(
            remaining_weight.saturating_sub(<T as Config>::WeightInfo::on_idle()),
        );

        if meter.limit().is_zero() {
            return Weight::zero();
        }

        SessionEndings::<T>::mutate(|endings : &mut Vec<SessionEnding<T>> | {
            loop {
                if endings.is_empty() {
                    break;
                }
```

6. MaximumBoundValidators is not an enforced maximum, and hence the maximum weight calculated here is not necessarily accurate. We recommend replacing the type of TotalValidatorStaked by a CountedStorageNMap to measure exactly how many elements will be processed.
7. The process of rewarding a contract does not depend on how many NFTs the contract has. Actually, when inspecting the corresponding weights, we can see that having more or less NFTs attached does not influence the resulting weight. Hence, we recommend removing the parameter from the benchmark and the resulting WeightInfo trait. The benchmark should be refactored so that it always has *MAX_NFTS_PER_CONTRACT* attached.
8. In the reward_contract function, we think it is more intuitive to pass the *total_v_imbalance* variable as a mutable reference, and get it modified during the process, similar to what occurs with the *context* variable. The function should hence not return a balance.

9. There are several places throughout the code where raw mathematical operations are performed instead of checked or saturated ones. We recommend replacing them with the latter to avoid unexpected side effects.
10. Consider whether slashing could be the first step being performed, instead of delivering rewards. The reason is that slashing should, if possible, be done as soon as possible because from the chain's stability perspective it is more optimal to chill those validators with two consecutive faults as soon as possible rather than delivering the rewards to the rest of the validators. In a normal scenario there should not be a big difference, but in certain scenarios the chain may be subject to a high workload that could prevent the *on_idle* hook to be granted with sufficient remaining weight to do these operations fast enough during a reasonably large amount of blocks.
11. This debug log entry should be of type error and include relevant information for developers to figure out the source of the discrepancy for debugging purposes.
12. There are several hooks that intervene during the process. While the weight is actually measured when running the benchmarks, the pallet itself is not in control of the logic inside of those. If the hooks' logic has several different execution paths depending on the current state, benchmarks may not reflect the real weight consumed. There are different non-exclusive options to alleviate this:
    - Ensuring or at least recommending developers implementing this trait to ensure there is only one deterministic logic workflow that can be consistently measured. If these hooks can actually consume more weight that what was measured when running the benchmarks there is a risk that *on_idle* exhausts the resources.
    - As a last-resource workaround, *on_idle* could be limited not to the passed remaining whole weight, but to the difference between that value and say 5-10% of the total block weight. This way there is room for inaccuracies during the execution that may not yield block exhaustion, obviously at the cost of not using the full capacity of a block.
13. There is an *unwrap* operation being performed. In an execution workflow where there is no *permission_nft* value the chain will stall. We highly recommend handling the case where such value is not present. **This is a major issue.**
14. Similarly, there are several *expected* statements throughout the codebase. We recommend addressing them all. **This is a major issue.** Particularly:
    - Slashing:
        - Case 1
        - Case 2
    - Rewards:
        - Case 1
        - Case 2
        - Case 3

    - Case 4
15. We recommend placing a *noop* statement in this line to indicate the compiler that the code should not be tested. When testing the pallet there is a failing test generated out of this snippet that fails.
16. There are two cases where the current staging layer is read yet not used:
    - Case 1
    - Case 2
17. Based on the HTML test coverage report, there are no tests that cover the case of the session ending process being stalled. If that is actually the case, we recommend adding a test that covers that part and checks if the process can be gracefully resumed. To find the test coverage, run the following command: `cargo llvm-cov --html -p pallet-nft-staking`
18. Due to the nature of the *on_idle* hook, and given that it is never known when it can execute or how much weight it will have available, there exists the possibility that there are several blocks where normal operations that mutate the state are being performed while the session is ending. The mutation of these storage items may impact how the precious session should end, and hence it is crucial to ensure there are no unintended modifications of the state that could translate into error in the session ending logic. Since there is no 100% certainty that a session will end during the next one (although very unlikely to happen) we have to **start from the premise that there exists the possibility that there are several sessions queued to be ended.**
    In this regard, we have identified and recommend to review suspicious usages of the following storage items outside of the session ending logic:
    - *TotalValidatorStakes*
        - When unbinding a validator.
        - When getting the current total stake.
        - When getting the current stagged total validator stake.
        - When setting the total validator stake in staging.
    - The *CurrentStagingLayer* rotates between sessions in the commit stage. However, in case a session could not be fully ended during the next one, the new session would start using the old one. We suspect this could be an issue that would invalidate having more than one queued session. Similarly to this, *Contracts* and *TotalValidatorStakes* are also stored per layer and are subject to the same situation.
    - UnlockingCurrency
        - When unstaking.
        - When kicking.
    - *UnlockingDelegatorNfts*
        - When kicking.
        - When unstaking.
19. We want to raise awareness about the fact that **there is absolutely no guarantee that *on_idle* will be executed with a certain periodicity and**

sufficient weight, and it cannot be safely assumed the tasks contained in it will be executed with a relative frequency that aligns with the chain's stability. In this regard, and related to the previous point: in case there could be issues now or in the future of this pallet's development that may overcomplicate things, we propose an alternative approach: place the current logic in the *on_initialize* hook. You can compute the remaining weight in the block, and **return the full block weight** if the computation was not finished. **This implies no new extrinsics will be allowed for a few blocks until the session ending process concludes.** Here is a sample implementation illustrating this:

```rust
#[pallet::hooks]
impl<T: Config> Hooks<BlockNumberFor<T>> for Pallet<T> {
    fn on_initialize(_block_number: BlockNumberFor<T>) -> Weight {
        let used_weight : Weight = frame_system::Pallet::<T>::block_weight().total();
        let max_block : Weight = T::BlockWeights::get().max_block;
        let remaining_weight : Weight = max_block.saturating_sub(used_weight);

        let mut meter : WeightMeter = WeightMeter::with_limit(
            remaining_weight.saturating_sub(<T as Config>::WeightInfo::on_idle()),
        );

        // ---> This should be refactored so that there is only one session, not a list of them.
        SessionEndings::<T>::mutate(|endings : &mut Vec<SessionEnding<T>> | {
            // ---> The loop is gone, as we only process one session.
            if endings.is_empty() {
                return Zero::zero();
            }

            // `SessionEnding` is not `Clone` and must be consumed.
            let SessionEnding { state : State<T> , mut context : Context<T> } = endings.remove( index: 0);

            match state.try_complete(&mut context, &mut meter) {
                // `SessionEnding` completed, try progressing the next one
                Ok(()) => {
                    Self::deposit_event( event: Event::<T>::SessionEndingCompleted {
                        session: context.session_index,
                    });

                    // ---> All good. We can simply return what was actually consumed.
                    // ---> The rest of the extrinsics can now be included.
                    meter.consumed()
                },

                // `SessionEnding` made progress, continue next time
                Err(state : State<T> ) => {
                    // Place new state into the previous' place
                    endings.insert( index: 0,  element: SessionEnding { state, context });

                    // ---> We could not finish the process, so we leave no more room+ for extrinsics.
                    remaining_weight
                },
            }
        })
    }
}
```

Grading:

Scope Item 1: Session ending

Topic	Grade	Feedback
Documentation	2	The code should be further documented.
Code Structure	4	Code structure is overall correct. Minor adjustments could be made to improve it.
Code Performance	4	Code performance is mostly correct, although there are a few corner cases where it could be further improved.
Tests	2	While most of the logic has sufficient test coverage, there is no single test that covers the case where the rewarding system gets stalled and must resume.

Supporting Reference for Assessment & Grading:

Criteria:

a) **Deliverable Functionality:** Assess whether the delivered code fulfills its intended purpose, a crucial aspect of evaluation.
b) **Documentation Quality:** Evaluate the comprehensiveness and clarity of documentation accompanying the delivery, aiming for accessibility to the broader community.
c) **Code Structure and Readability:** Examine the layout and clarity of the codebase to ensure it is understandable for community members who may review it.
d) **Performance Optimization:** Investigate the efficiency and performance of the chain, emphasizing avoidance of redundancies and straightforward implementation without the need for extensive benchmarking.
e) **Testing Coverage:** Emphasize the importance of thorough testing by evaluating the extent to which the project incorporates comprehensive testing protocols.

Grading: Scoring 1-5.

Documentation:

- **1) Inadequate:** Absence of documentation.
- **2) Deficient:** Documentation is present but lacks essential components.
- **3) Fair:** Existing documentation poses challenges in comprehension.
- **4) Adequate:** Documentation is comprehensive yet demands moderate effort to grasp.
- **5) Exemplary:** Documentation is impeccably structured, comprehensive, and easily comprehensible.

Code Structure:

- **1) Inadequate:** Code structure is convoluted and difficult to follow.
- **2) Deficient:** Some attempt at segmentation based on objectives, but lacks overall coherence.
- **3) Satisfactory:** Broad segmentation exists, however, with deficiencies in detailed structuring.
- **4) Proficient:** Segmentation of processes and tasks demonstrates good practice, albeit with minor shortcomings.
- **5) Exceptional:** Code division and structure are flawless, facilitating ease of comprehension and replication.

Code Performance:

- **1) Deficient:** Code performance is hindered by extensive repetitive patterns and lack of segmentation.

- **2) Subpar:** Notable redundancy exists, particularly within complex sections of the codebase.
- **3) Acceptable:** Segmentation is apparent, with discernible efforts to minimize redundancy, albeit not fully achieved.
- **4) Competent:** Overall execution is commendable, with minor areas identified for optimization.
- **5) Outstanding:** Code execution demonstrates exceptional performance, characterized by minimal redundancy and optimal efficiency.

Tests:

- **1) Deficient:** Testing procedures are entirely absent.
- **2) Limited:** Minimal testing efforts are evident, often simplistic in nature.
- **3) Adequate:** Testing coverage is satisfactory, though certain scenarios remain untested.
- **4) Proficient:** Testing protocols cover a majority of components adequately, with some areas identified for enhancement.
- **5) Exemplary:** Comprehensive testing which encompasses all conceivable scenarios, reflecting exemplary adherence to testing standards.

Substrate Code Review Feedback

Prepared By: BlockDeep Labs
Prepared For: Mosaic

Team: Mosaic
Review Date: 09-06-2025
Report Date: 17-06-2025

Commit Hash: 87f9f2d51e2223d471c2fd8c9321d7737b483830
Review Scope Items:

Assessment:

Documentation and Codebase Quality:	**Feedback:**
a. Is there sufficient documentation for compiling and executing the code?	The documentation to assist compilation is sufficient.
b. Does the project documentation provide a conceptual understanding of the project?	Yes, the project includes basic documentation, providing an overview but it's not exhaustive. There is room for improvement.
c. Are there written tests for the codebase, and do they run successfully, indicating completed code components?	There are written tests and they run successfully, providing a decent test coverage.

Code Review Comments:

Airdrop Pallet - Authentication Mechanism

Customer's Initial Status and Concerns

The customer implemented an airdrop mechanism using an unsigned extrinsic whose payload is signed by their hot wallet as an authoritative origin. Their goal with using unsigned transactions was greater control over the priority of the airdrop extrinsics, and to limit the number of such calls in a single block. Unfortunately, it required a nonce to strictly order the extrinsics coming from the hot wallet that mints their presold NFTs and tokens from the external sales DB onto the chain.

Keeping that nonce in storage causes unintuitive wallet integration, because when the extrinsic errors, that storage is rolled back together with all other storage changes. This makes it hard to process multiple requests in parallel, and requires the wallet to wait until a call is finalized before sending in the next one.

The customer's primary concern was the operational inefficiency: the nonce system forces the hot wallet to wait for previous transactions to complete before submitting the next one, creating a significant bottleneck that prevents parallel processing of multiple airdrop requests.

Problems found with the implemented nonce mechanism

After analyzing the airdrop pallet implementation, we confirm the validation logic in validate_unsigned enforces strict ordering where transactions must be processed in exact nonce order. This confirms the sequential bottleneck issue, preventing parallel processing and creating operational inefficiencies.

Validation Issues Identified

- **Fee Bypass Without Rate Limiting**: The unsigned extrinsic mechanism bypasses all fee payments, but there's no built-in rate limiting beyond the nonce system. An attacker with access to valid signatures could potentially spam the network.
- **Mempool Misuse**: Valid and properly signed transactions with future nonces cannot remain in the mempool until the chain has processed the previous nonce.

Furthermore, the mere existence of a centralized hot wallet arises several concerns:

Centralized Hot Wallet Authority Issues

- **Single Point of Failure**: The entire airdrop system depends on one hot wallet as the minting authority, creating catastrophic failure risk if the wallet is compromised, becomes unavailable, or experiences operational issues.
- **Security Vulnerability**: Hot wallets are constantly online and exposed to network-based attacks, malware, and system compromises, making the private key vulnerable to theft or misuse.
- **Centralization Risk**: One entity controls unlimited token minting, NFT creation, and validator permission distribution, undermining blockchain decentralization principles and enabling potential economic manipulation.

Proposed Solution

Validation Checks

We suggest replacing the unsigned extrinsic with a normal signed transaction from a predefined authenticated origin, returning Pays::No to avoid transaction fees in case of success, and specially penalizing attackers by charging a fee.

This proposal has the following advantages over the existing solution:

- **Removes sequential bottleneck**: Normal transaction pool ordering can be used.
- **Maintains fee exemption**: Pays::No preserves the no-fee requirement.
- **Better spam protection**: Standard transaction validation and rate limiting apply.
- **Cleaner mempool**: Failed transactions are rejected rather than processed, and the corresponding fee is subtracted from the sender account.

- **Transaction priority achieved**: by sending a tip, the extrinsic may be included faster in the chain.
- **Parallelization achieved**: Several transactions can be included in the same block, substantially increasing the throughput.

Also note that we do not recommend limiting calls to the airdrop extrinsic. We understand that the idea is that transactions other than airdrop claiming keep properly functioning in times of stress, but Substrate-based chains already implement a tipping system to incentivize block producers to include these extrinsics, and that is the mechanism that should be used in case the network is congested. Besides, looking at the maximum configured weight of the parachain's block and the extrinsic's measured one, theoretically around 384 airdrop extrinsics could fit per block in the worst case, 514 in the average case, and 774 in the best case. That should minimize the potential downtime the chain may suffer during spikes.

Here is the test used to perform these calculations:

```
#[test]
fn get_block_proportion() {
      let max = frame_system::limits::BlockWeights::with_sensible_defaults(
            Weight::from_parts(2u64 * 1_000_000_000_000, u64::MAX),
            Perbill::from_percent(75),
      ).max_block;
      // Change between 5, 3 and 1 for the worst, average and best cases.
      let base = <Test as Config>::WeightInfo::airdrop(5);

      println!("ref_time = {:?} | proof_size = {:?}", max.ref_time() /
base.ref_time(), max.proof_size() / base.proof_size());
}
```

Implementation Considerations:

- The predefined origin's account would need sufficient balance, but it would be just a little higher than the existential deposit.
- Root or council governance could manage the authorized origin.
- The dispatch info should return Pays::No to maintain fee exemption in case of success. See here for an example with the sudo pallet in Polkadot-SDK.
- Standard transaction priority and ordering mechanisms would apply by paying a tip.
- No maximum number of calls in a single block to the airdrop extrinsic.

Security Improvements:

- Provides better transaction ordering flexibility.

- Maintains cryptographic authentication while improving DoS resistance
- Leverages existing Substrate transaction validation infrastructure.
- Adheres to Substrate's best practices.

Alternative Approach to Hot Wallet Usage

Instead of a centralized approach using hot wallets, airdrops could also be implemented using Merkle trees, which are cryptographic data structures that organize recipient data into a binary tree format. In a Merkle tree based airdrop approach, each leaf node represents a recipient's allocation data (such as wallet address and token amount), while each internal node contains the cryptographic hash of its children nodes, ultimately culminating in a single root hash that cryptographically commits to the entire distribution. This structure provides several key properties: efficient verification (recipients only need to store their leaf data and the path to the root rather than the entire dataset), space efficiency (only the root hash needs to be stored on-chain regardless of distribution size), and tamper resistance. That implies any modification to recipient data fundamentally changes the root hash, making unauthorized alterations immediately detectable.

The potential presence of Merkle proof validation inside the airdrop pallet does not pose a significant calculation overhead, as the number of operations performed scales logarithmically with the number of nodes in the tree.

This approach has the following advantages:

- Provides maximum security by keeping the signing key offline after initial deployment.
- Offers significantly better user experience by eliminating the need for customers to manage complex claim codes or pre-register wallet addresses during signup.
- Ensures cryptographic proof of entitlement based on existing customer records.
- Prevents unauthorized claims through email verification.
- Allows flexible address selection during the claiming process while maintaining full audit trails of all distributions through blockchain transparency.

For further inspiration, see the following relevant projects using the Merkle Tree based airdrop approach:

- Uniswap
- Gitcoin

Airdrop Pallet - Feedback on Best Practices

1. The pallet lacks in-code documentation. We recommend adding comprehensive documentation throughout the codebase to help readers understand the pallet's purpose as a whole and for each of its components.
2. The pallet is tightly bound to a particular signature schema (SR25519 in this case. This should be configurable via pallet-wise configuration item by adding the functionality to check the signature. You can check FRAME's NFT pallet for inspiration.
3. The minting_authority function uses expect, which can stall the chain if the storage is not initialized. We recommend returning an error instead so that changes are rolled back.

```
fn minting_authority() -> Result<sr25519::Public, Error<T>> {
    MintingAuthority::<T>::get().ok_or(Error::<T>::NotInitialized)
}
```

4. The MAX_DELEGATOR_NFTS constant is fixed. We recommend using runtime-defined constants in the pallet's configuration trait for added flexibility.
5. Potential overflow and underflow in the priority calculation logic. We recommend using saturating additions and subtractions.
6. Something similar occurs in the balance minting logic. Furthermore, in this case we also recommend making the logic more explicit and add comments to better understand such a critical operation:

```
// Calculate how much additional balance is needed to reach minimum balance
let total_after_endowment = current_balance.saturating_add(endowed_balance);
let additional_for_ed = minimum_balance.saturating_sub(total_after_endowment);
let total_to_mint = endowed_balance.saturating_add(additional_for_ed);
```

7. There is another underflow issue in the validate_unsigned function.
8. The logic does not check that the user has the existential deposit covered before minting NFTs. There is a balance being minted, but afterwards it would be a good practice to include a defensive check to ensure that the NFT can actually be minted by comparing the user's balance against the existential deposit.
9. Benchmarks should be tested as well.
10. Benchmarks do not compile. It is crucial for the chain to properly measure the weight of all operations. See here for more details.

Grading:

Scope Item 1: Airdrop Pallet

Topic	Grade	Feedback
Documentation	3	The pallet has a good README, but lacks in-code documentation.
Code Structure	4	Code structure is overall correct. Minor adjustments could be made to improve it.
Code Performance	3	Code performance is mostly correct, although there are a few corner cases where it could be further improved. Benchmark logic is correct, but It is crucial that benchmarks can be successfully compiled and executed to obtain the weights.
Tests	4	The test coverage is sufficiently exhaustive, although it could be improved with further checks and tests. We recommend obtaining the per-line coverage and aiming to reach near 100% line coverage. Furthermore, consider adding comprehensive tests for edge cases like integer overflow, invalid signatures, and boundary conditions.

Supporting Reference for Assessment & Grading:

Criteria:

a) **Deliverable Functionality:** Assess whether the delivered code fulfills its intended purpose, a crucial aspect of evaluation.
b) **Documentation Quality:** Evaluate the comprehensiveness and clarity of documentation accompanying the delivery, aiming for accessibility to the broader community.
c) **Code Structure and Readability:** Examine the layout and clarity of the codebase to ensure it is understandable for community members who may review it.
d) **Performance Optimization:** Investigate the efficiency and performance of the chain, emphasizing avoidance of redundancies and straightforward implementation without the need for extensive benchmarking.
e) **Testing Coverage:** Emphasize the importance of thorough testing by evaluating the extent to which the project incorporates comprehensive testing protocols.

Grading: Scoring 1-5.

Documentation:

- **1) Inadequate:** Absence of documentation.
- **2) Deficient:** Documentation is present but lacks essential components.
- **3) Fair:** Existing documentation poses challenges in comprehension.
- **4) Adequate:** Documentation is comprehensive yet demands moderate effort to grasp.
- **5) Exemplary:** Documentation is impeccably structured, comprehensive, and easily comprehensible.

Code Structure:

- **1) Inadequate:** Code structure is convoluted and difficult to follow.
- **2) Deficient:** Some attempt at segmentation based on objectives, but lacks overall coherence.
- **3) Satisfactory:** Broad segmentation exists, however, with deficiencies in detailed structuring.
- **4) Proficient:** Segmentation of processes and tasks demonstrates good practice, albeit with minor shortcomings.
- **5) Exceptional:** Code division and structure are flawless, facilitating ease of comprehension and replication.

Code Performance:

- **1) Deficient:** Code performance is hindered by extensive repetitive patterns and lack of segmentation.

- **2) Subpar:** Notable redundancy exists, particularly within complex sections of the codebase.
- **3) Acceptable:** Segmentation is apparent, with discernible efforts to minimize redundancy, albeit not fully achieved.
- **4) Competent:** Overall execution is commendable, with minor areas identified for optimization.
- **5) Outstanding:** Code execution demonstrates exceptional performance, characterized by minimal redundancy and optimal efficiency.

Tests:

- **1) Deficient:** Testing procedures are entirely absent.
- **2) Limited:** Minimal testing efforts are evident, often simplistic in nature.
- **3) Adequate:** Testing coverage is satisfactory, though certain scenarios remain untested.
- **4) Proficient:** Testing protocols cover a majority of components adequately, with some areas identified for enhancement.
- **5) Exemplary:** Comprehensive testing which encompasses all conceivable scenarios, reflecting exemplary adherence to testing standards.