Learn ink! by Example Part 2 - Upgrade ink! Smart Contracts

Smart contracts deployed on public blockchains are mostly irrecoverable because of immutability, unless they are upgradable. Upgradable smart contracts are critical for security patches, important bug fixes, and normal functional changes. Given that most blockchain-based applications are implemented in smart contracts (e.g. deFi, DEX, bridges, rollups, NFTs, gaming, social, etc.), it is essential to understand what it takes to seamlessly upgrade smart contracts.

We published the Medium article "Learn ink! by Example - Order Food on the Blockchain" a while back, sharing with the community how to leverage macros in ink! smart contracts for code simplicity, reusability, and readability. This is part two of the "Learn ink! By Example" series, focusing on ink! smart contract upgradability.

Smart contract upgradability challenges

There have been challenges in smart contract upgradability, such as:

1. Contract storage collisions and corruption: When upgrading a contract, it is critical to ensure that the new contract code does not overwrite the data storage of the old code.

Persisted contract data can enable transparent upgrades without experiencing disruptions.
2. Interoperability and security risks: Whenever a contract is upgraded, there is a risk that the new contract code could introduce security vulnerabilities or fail to seamlessly interoperate with the context of the old contract.
3. Cost: Upgrading a contract can be a costly operation, both in terms of gas and DevOps time, as the new contract needs to be deployed with all the contract's state for continuity.

Ink! is a smart contract language that allows developers to write smart contracts that can be upgraded with flexibility and storage control. This is a valuable feature as it enables developers to fix bugs and improve the functionality of their contracts without having to create a new contract from scratch.

Understand ink! smart contract storage layout

Ink!'s smart contract storage organizes itself as a key/value database that is exposed to ink! by the contracts pallet. The storage API operates by storing and loading entries into and from storage cells. Each storage cell is accessed under its own dedicated storage key.

Packed vs non-Packed storage layout

Types that can be stored entirely in a single storage cell are considered Packed. These include bool, which represents a boolean value (true or false); u8, u16, u32, u64, and u128, which are unsigned integer types of various bit sizes; i8, i16, i32, i64, and i128, which are signed integer types of various bit sizes; char, which represents a Unicode character; Address, which represents an account address on the blockchain; Balance, which represents the balance of an account in the native currency; and Timestamp, which represents a point in time.

On the other hand, there are data types, such as ink::prelude::vec::Vec, that may exceed the storage cell boundary. Additionally, there is a constraint on the capacity of the buffer used for encoding and decoding storage items, which is limited to around 16KB in the default configuration. This means that any contract attempting to decode a larger size than that will result in an error. This is where non-Packed storage layout comes into play, particularly for types like String, which represents a dynamic string of characters; Vec<T>, which represents a dynamic array of elements of type T; and Option<T>, which represents an optional value that can be either Some(value) or None. Finally, Result<T, E> represents a result that can be either Ok(value) or Err(error).

Eager loading vs lazy loading

Ink! provides the Lazy primitive to break up the storage into smaller pieces, which can be loaded on demand. Wrapping any storage field inside a Lazy struct makes the enclosing storage struct non-Packed, preventing it from being eagerly loaded during storage operations. When loading data from storage, ink! smart contracts can either eagerly load all the data at once or lazily load only the data that is needed.

Any message interacting with a Packed storage layout will always need to operate on the entire contract storage struct. In comparison, with a non-Packed storage layout, the access could be piecemeal for specific storage items on demand. So, depending on the contract data structure and access patterns, there are tradeoffs in designing contract data storage, using Packed vs non-Packed storage layout at optimal granularity, eager loading vs lazy loading, etc. For more details, please refer to this ink! Document.

layout-ful vs layout-less

Ink! makes it easier to upgrade smart contracts without having to migrate the data or interrupt the user experience through ink!'s flexible storage layout design, which accommodates different data types and access patterns.

A smart contract data field is layout-less if its type is Lazy, Mapping, or layout-less types. A data type is layout-less if its fields are layout-less; otherwise, the field is layout-ful. All Packed types are layout-ful by default, but not every non-Packed type is layout-less.

The order of layout-less fields can be changed or removed without problems in ink! smart contracts. However, layout-ful fields cannot be. Making each field layout-less allows you to add/remove/reorder fields with a reduced chance of storage layout corruption. However, doing this for every storage field may not be optimally performant depending on use cases. Fields expected to be updated in the future should be layout-less. The fields expected to remain through upgrades or be dropped later can be layout-ful. It is important to understand ink! smart contract upgradability behaviors.

Manual vs automatic storage key generation

Ink! provides both the AutoKey primitive, which auto-generates keys during contract compile time, and the ManualKey primitive, which allows manual control over the storage key for non-Packed types. Whether AutoKey or ManualKey is used does not affect the field ordering, as the auto storage key is generated based on the field's name rather than its position.

AutoKey is convenient, but there is no guarantee that the key calculation algorithm will remain the same in future versions of ink!. ManualKey allows developers to set a persistent key value throughout the lifecycle of the smart contract. Use ManualKey if you plan to rename the field in the future.

OpenBrush provides the #[openbrush::storage_item] attribute macro, which implements traits for a struct. It also auto-generates unique storage keys for each of the struct's fields that are marked as #[lazy], Mapping, or MultiMapping. The macro-generated storage keys are Consts with the pattern STORAGE_KEY_{struct_name}_{field_name} in the generated code.

In terms of the differences between the storage_item and storage_unique_key macros, the #[openbrush::storage_item] macro is an attribute macro used for all members of the applicable data struct, with a dedicated storage key generated for each field of that given struct.

OpenBrush also provides the openbrush::storage_unique_key! macro, which generates a storage key based on the path to the structure. The storage_unique_key! macro is a declarative macro that generates a unique storage key using parameters provided for manual key generation.

Each logic unit in ink! smart contract code should have a unique storage key and should only be used once in the same contract. Here's an example of how we defined the struct Data with the storage_item macro in the data.rs file.

```
#[openbrush::storage_item]
pub struct Data {
    pub food_id: FoodId,
    pub food_data: Mapping<FoodId, Food>,
    …
}
```

What could cause contract storage collisions and corruptions?

Storage collisions and corruptions scenarios

Based on the above understanding of ink! storage layouts and nuances, developers can better understand the causes of ink! smart contract storage collisions and corruptions. Some culprits include, but are not limited to:

- Incompatible storage layouts: When upgrading an ink! smart contract, the new contract may have an inconsistent storage layout, which could corrupt the contract's data storage.
- Adding, removing, or reordering layout-ful fields.
- Future ink! versions may change the storage key generation algorithm.
- Different serialization and deserialization mechanisms used for storage cell reads/writes, especially if they are different before and after upgrades.
- Using the same type in several fields can cause a collision on the same storage key if the

<ParentKey: StorageKey = AutoKey> is not specified during the declaration of the Child type and the ParentKey is not passed into ManualKey. For details about several corner cases involving nested parent-child structs, please read this StackExchange page.

ink! smart contract upgrade best practices

Developers should be aware of the challenges involved in upgrading Ink! smart contracts and take steps to mitigate these risks. To avoid these risks, it is important to plan and execute any upgrades to ink! smart contracts carefully.

To mitigate the risks of smart contract storage collision and corruption, developers should:

- Carefully review the data structs and storage layouts to ensure compatibility.
- Thoroughly test the new code against data storage before deployment.
- Monitor the contract after upgrades to identify any potential problems.
- Develop smart contract upgrade tools to detect potential storage collisions/corruptions as a utility.
- View the Substrate/Polkadot SDK runtime as the state transition function of the blockchain, which comprises the data type definitions. To ensure seamless operation before and after runtime storage migration, the storage and runtime logic need to work together.
- Note that Substrate runtime storage migration is out of scope for this article. Please refer to the Substrate tutorial for migrations for specific pallets or individual storage items.
- Consider smart contract auditing and bug bounties to minimize risks.

To keep ink! smart contracts lean and avoid bloat, developers should:

- Remove data fields that are no longer used after careful validation without causing corruptions.
- Continuously optimize contract storage to adapt to changing application needs.
- Perform storage migrations in response to storage changes in upstream pallets.
- Simplify the design of contracts by splitting logic across multiple contracts and designing data structures with consistency and composability to accommodate future changes and survive upgrades.
- Reduce complexity by carefully planning storage design based on data access patterns, prioritizing data continuity, and keeping data extensible for future changes.
- Keep in mind that while an ink! smart contract theoretically has almost no limit, there are gas costs and the block gas limit. Therefore, it is good practice to keep storage to a minimum to avoid bloating the storage footprint, for simplicity, upgradability, and gas savings.

Follow the best practice rules in the Upgradable contract document to minimize the chance of storage corruption/collisions during smart contract upgrades.

Upgrade pattern recommendation and comparisons

ink! smart contract upgrade patterns

This ink! document outlines three smart contract upgrade patterns, each with its own pros and cons.

- Proxy pattern
- Usage of the set_code_hash method
- Diamond standard pattern

The proxy pattern was introduced before ink! had the set_code_hash function available. The proxy contract provides contract storage and delegates calls to other contracts.

The proxy contract uses delegate_call to forward calls from the proxy contract to the new proxied contract. The code in the new proxied contract needs to work in the context of the proxy contract. This might be a source of security exploits. For example, Solidity smart contract upgrade uses the proxy pattern where the proxy contract dispatches all function calls to the logic contract using the delegatecall() function. The logic contract's code is executed in the context of the calling proxy contract. Instead of writing to its own storage, the logic contract writes to the calling contract's storage. This increases the chances of introducing bugs or vulnerabilities, which in turn increases security risks.

Strictly speaking, the diamond pattern is more about contract code structuring across multiple logic layers, substituting diamond facets with contracts of new behaviors. Since an ink! contract has five times the space, so it is often unnecessary to use the diamond pattern.

The diamond pattern is not exactly an upgradeability pattern, but more of a code structuring pattern. The upgradeability lies in the way you substitute the facets of the diamond contract in order to change the contracts' behavior. Here again, opposed to Solidity, you have five times the space for your contract, so the diamond pattern is unnecessary most of the time.

This leaves the ink! native set_code_hash pattern, which calls the set_code_hash method to point to a different on-chain code hash. This is possible because the contract code hash is stored in the contract's state and separated from the contract's code. Compared with the above two patterns, this approach only requires one contract before and after upgrades. Messages continue to interact with the same contract address with storage continuity. Practically, this is the recommended upgrade pattern for ink! upgrades, as the proxy pattern is likely to be deprecated in the future.

Solidity smart contract upgrades comparisons

In Solidity, the proxy pattern is often used to achieve contract upgradability. The proxy contract delegates calls to a new contract. The ERC-1967 standardizes the storage slots used by the proxy to store the address of the logic contract they delegate to, as well as other proxy-specific information, reducing the likelihood of clashes with other contract storage.

In contrast, ink! contracts don't have a built-in proxy mechanism or a standard like ERC-1967. However, they can be made upgradeable through a similar pattern, where a proxy contract can delegate calls to a target contract. Notice that ink::env::CallFlags allow developers to control how a contract call is made. This includes options like REENTRANT and ALLOW_REENTRY, which can affect the behavior of upgradeable contracts. Here's an example.

Better yet, ink! leverages the native set_code_hash method to do an in-place contract code upgrade by pointing to the new on-chain code blob hash. ink! also supports dedicated storage keys to give developers more control over the smart contract storage via layout-ful vs layout-less design, eager vs lazy retrieval, auto vs manual storage key generation, to manage smart contract storage through upgrades with flexibility to add/remove/reorder data fields.

For Solidity, according to this Openzepplin document: "*Due to technical limitations, when you upgrade a contract to a new version, you cannot change the storage layout of that contract. This means that if you have already declared a state variable in your contract, you cannot remove it, change its type, or declare another variable before it. In our Box example, it means that we can only add new state variables after value*."

ink! generates storage keys based on the field's name, not on the position like in Solidity. While this provides flexibility to change data fields over upgrades and the possibility to scrub contract storage to avoid bloat, it also incurs potential risks if future ink! versions change the underlying key generation algorithms unless manual key generation is used to prevent possible collisions.

ink! code hash can point to Wasm (WebAssembly) on-chain code blob, while Solidity's EVM bytecode lives in the contract's account storage with only hash roots stored in blocks. Substrate enables runtime upgrades without hard fork as the runtime code blob is stored on-chain.

ink!'s upgrade pattern of set_code_hash plus dedicated storage keys starts and ends with the same contract and contract address. Solidity's upgrade pattern of proxy plus delegated call ends in additional proxied contracts besides the proxy base contract, which could cause proliferation of contracts if frequent upgrades.

Overall, ink! offers an upgradeability solution with more flexibility and control over contract storage, fewer contracts, and makes it possible for fork-less runtime upgrades.

Blockchain Food Order example

Business logic changes trigger upgrades

In Part 1 of this series, Learn ink! by Example - Order Food on the Blockchain, we explained the blockchain food order use case with personas and logic flow. The business logic is dynamic, requiring smart contract implementations to keep up with the dynamics.

For example, we introduced a new food delivery rule mandating that couriers can decline food deliveries over 50 miles. We added a random function to simulate delivery distance and changed the courier logic for distance-based delivery decline, thus requiring an ink! Smart contract upgrade.

The dApps are expected to keep existing data already in the smart contract storage, such as old restaurants, couriers, menu items, and historical orders, while continuing to support adding new data entries, such as new orders and new menu items, to keep the system running. This is called smart contract storage continuity. In the next section, we will walk through the major steps before and after upgrades.

For those who might be interested in the related code on GitHub, please check out the updated confirm order function, random number generation trait, random trait Implementation and foodorder/lib.rs. The set_code_hash function can be only called by the contract owner for security. We used both Ownable and Upgradeable like the following:

```
#[openbrush::implementation(Ownable, Upgradeable)]
#[openbrush::contract]
mod blockchainfoodorder {
 #[ink(storage)]
 #[derive(Storage, Default)]
 pub struct FoodOrder {
      #[storage_field]
      ownable: ownable::Data,
  }

...

// here's the smart contract constructor
impl FoodOrder {
       #[ink(constructor)]
       pub fn new() -> Self {
           let mut instance = Self::default();
           let caller = Self::env().caller();
           ownable::Internal::_init_with_owner(&mut instance, caller);
           instance
       }
   }
```

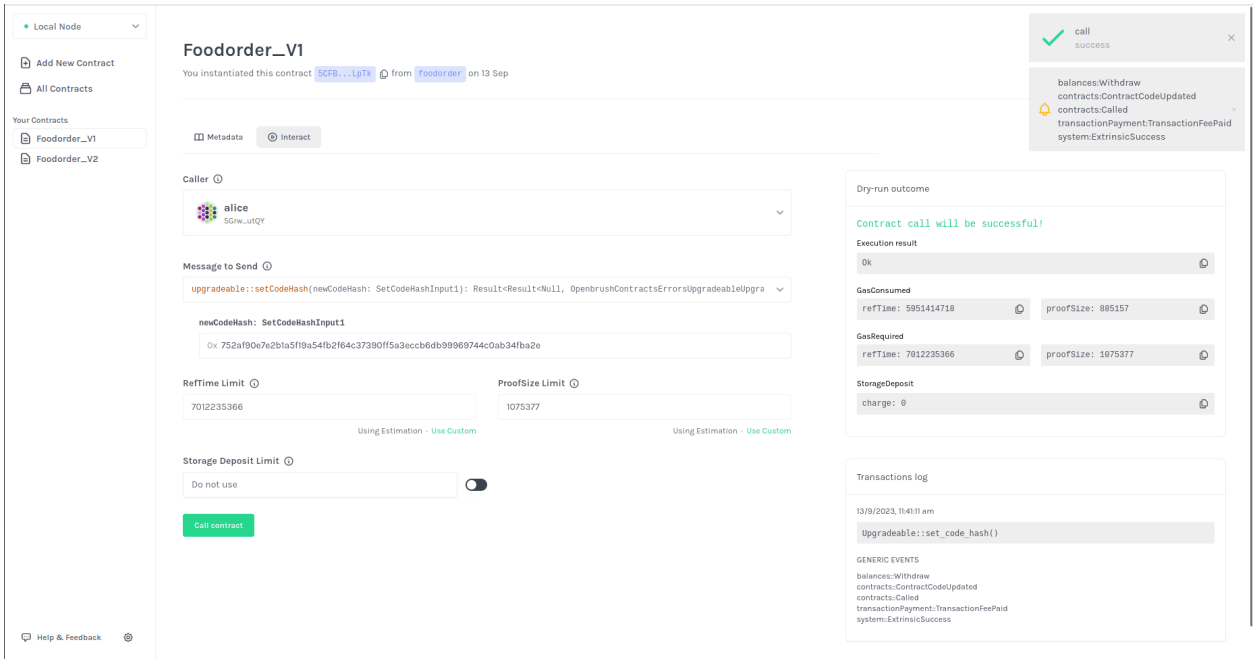
Here's the output from the test script including regression, upgrade step, and new tests post upgrade to confirm or decline an order based on the new business logic changes:

```
FoodOrder test
  Main Functionality
    ✓ Platform is ready
    ✓ Restaurant A is added (106ms)
    ✓ Courier A is added (140ms)
    ✓ Customer A is added (132ms)
    ✓ Food A is added (96ms)
    ✓ Order is submitted (120ms)
    ✓ Order is Confirmed (118ms)
    ✓ Food is cooked and Payment is transferred to restaurant (128ms)
    ✓ Order is Delivered (99ms)
    ✓ Delivery is accepted and Payment is sent to courier (189ms)

Upgradeable
  ✓ Upgrade with set_code_hash (374ms)
  ✓ Create new food (51ms)
  ✓ Submit new orders (161ms)
  ✓ Confirm an order (231ms)
  ✓ Decline an order (325ms)
  Check whether the old data remains
    ✓ Check the restaurant (66ms)
    ✓ Check the customer
    ✓ Check the order


18 passing (3s)
```

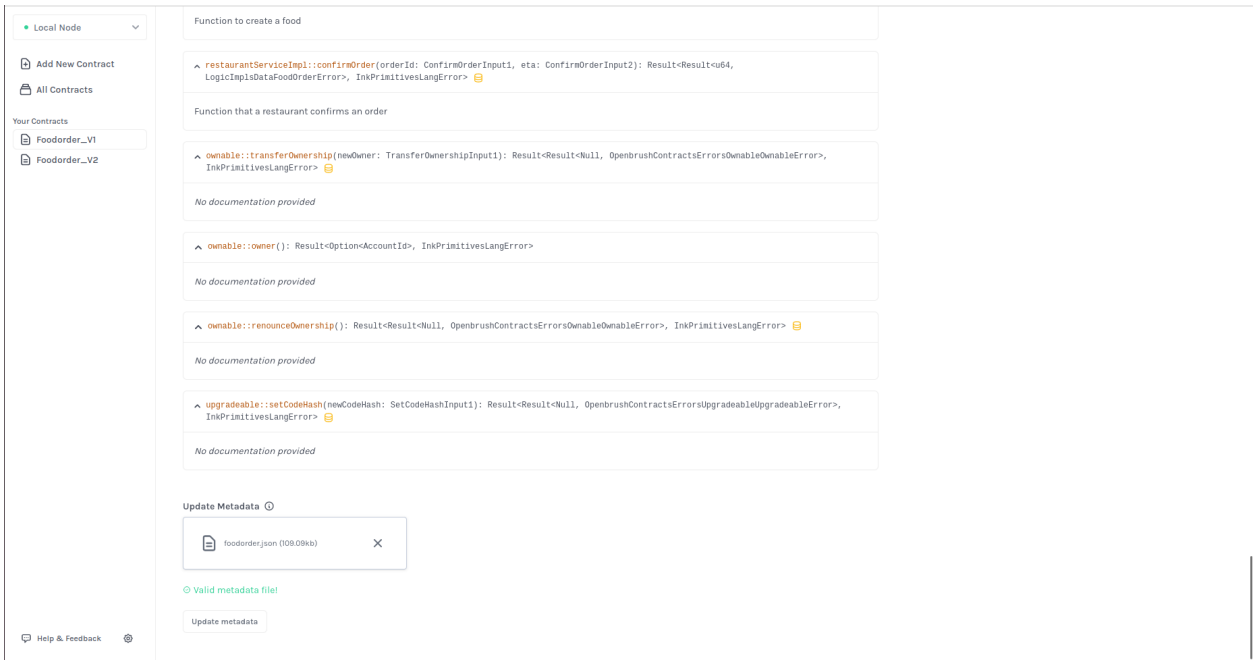
Upgrade the smart contract

In this Blockchain Food Order example, the FoodOrder_V1 ink! contract already has previously loaded data, such as restaurants and food items, delivery couriers, customers, orders, etc. FoodOrder_V1 emits a ConfirmOrderEvent every time a customer's order is received.

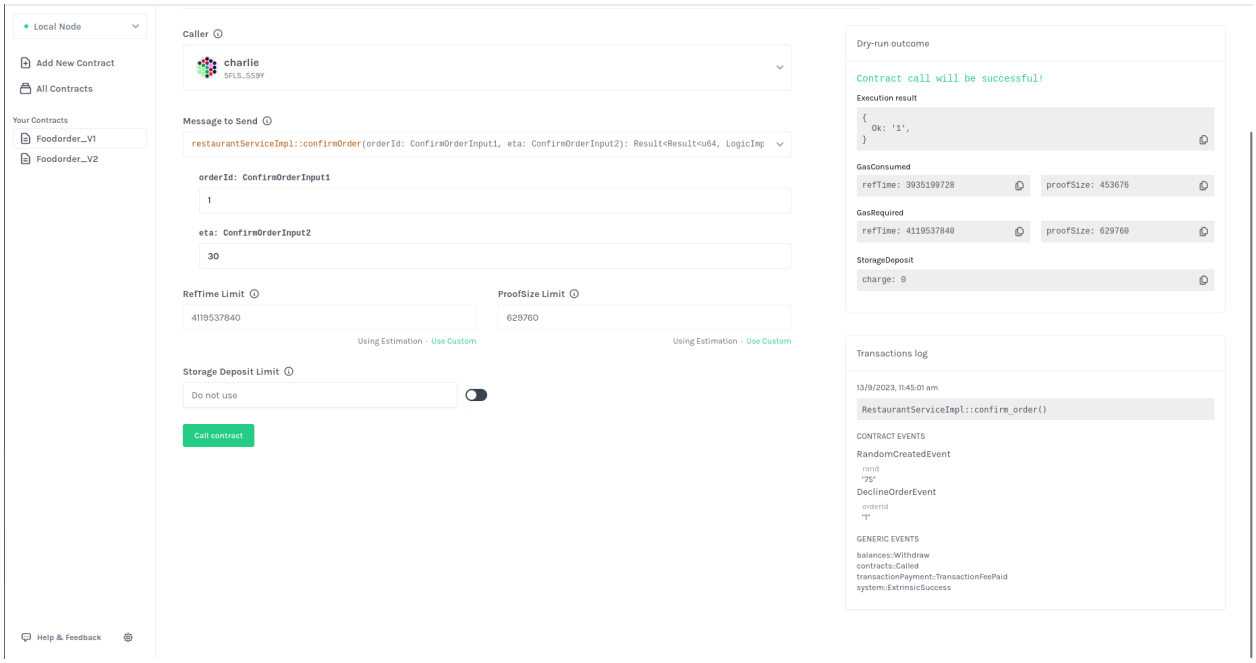
Now, you can upgrade it by calling the set_code_hash message of FoodOrder_V1 and setting the code hash of FoodOrder_V2.



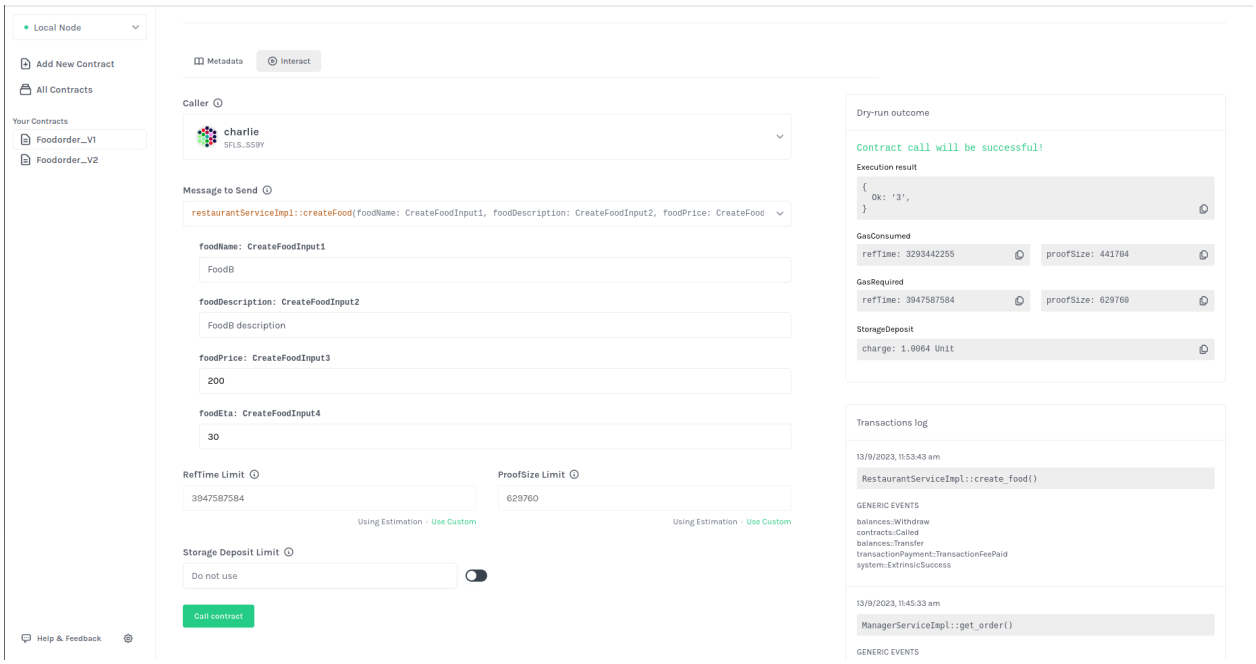
With the metadata updated from the foodorder.json file that was uploaded, which contains all the ABI interfaces and metadata generated from FoodOrder_V2.



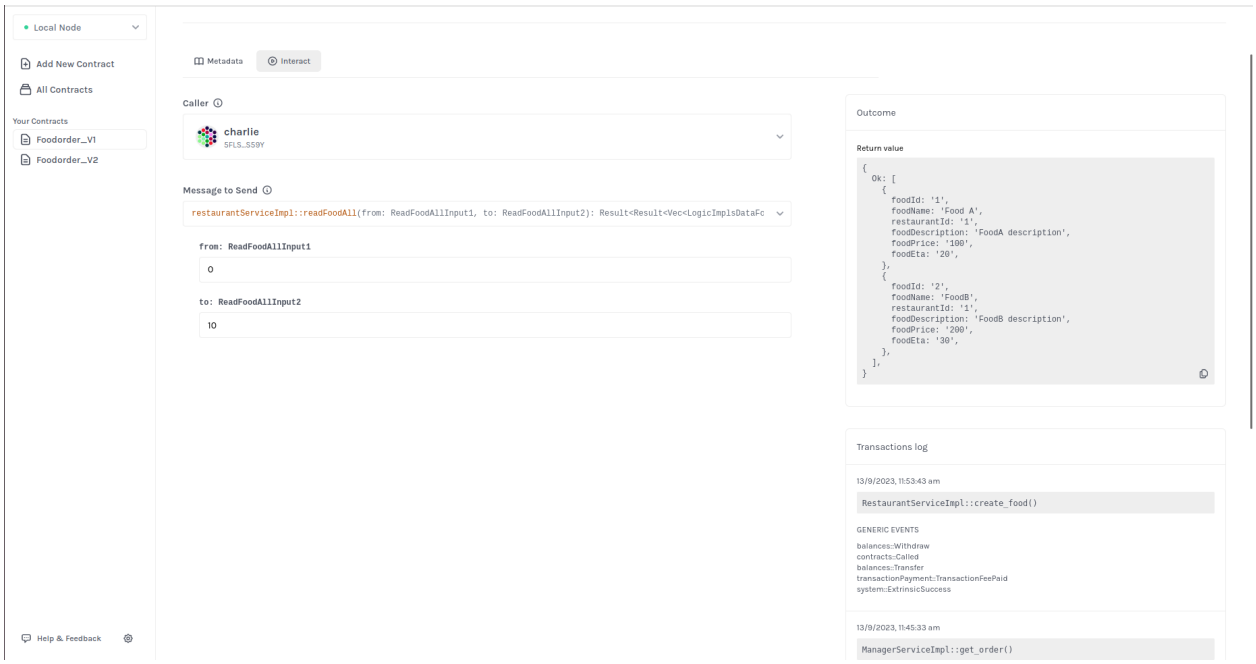
Based on the described changes in the business logic, FoodOrder_V2 generates a random mileage number and emits the RandomCreatedEvent upon receiving an order. The courier can choose to accept or decline the delivery if the distance is more than, let's say, 50 miles.



You can also add new data, such as new restaurant menu items, while maintaining the same user experience as before the upgrade.



To verify that the new food item was successfully added on top of the existing restaurant menu, you can invoke the read_food_all message to list them all.



The above screenshots demonstrate the ink! Contract upgrade, storage data continuity, and consistent user experiences during the upgrade.

For an end-to-end flow with screenshots, please refer to step-by-step guides in the project's GitHub docs directory, where there are detailed steps to run the contracts UI, set up against the testnet, get tokens from the faucet, and step-by-step screenshots on how to interact with the contract before and after the upgrade.

Suggestions on upgradability improvements

Just like typical software development, smart contracts have their share of bugs and risks. They are most vulnerable to potential attacks or falling apart due to storage corruptions during an upgrade. Applying smart contract-specific CI/CD (Continuous Integration / Continuous Development) processes to keep smart contract development and operational flow in check would also be instrumental for high quality and productivity. A robust simulation / dry run environment makes it easier to detect glitches during progressive development, testing, and deployment cycles. This is not specific to ink!, but general smart contract development.

While the ink! smart contract provides many advanced upgradability and storage layout features, it can be overwhelming for novice ink! developers. It would be helpful to develop a tool that can automatically check data type consistency and validate storage layout from old to new to catch potential upgrade-related storage collisions and corruptions.

The same ink! Upgrade Tool can also be extended to improve the diagnosis of existing smart

contract data lineage, track runtime pallet version dependencies, log events off-chain to monitor calls to smart contract functions and changes to state variables, audit contract behaviors before and after upgrades, and support smart contract governance for upgrade-related decisions. These compute-intensive tasks can be fulfilled via ink!'s OCW (Off Chain Workers) instead of on-chain for performance, efficiency, and cost-effectiveness.

For complex upgrades involving not just smart contracts but also runtime, multiple blocks can be in flight before the upgrade completes. Simulation with regression tests can help guard against known smart contract vulnerabilities such as reentrancy, oracle, bridges, etc. during the vulnerable upgrade period. In case of emergencies, the system should be capable of rolling back to restore pre-states.

Smart contract upgradability should be carefully planned and thoroughly tested along with Substrate and Polkadot SDK fork-less runtime upgrades, if any, to ensure seamless in-flight migration success. A smart contract lifecycle management tool would be really helpful to keep track of versioning, inter-dependencies, and performance monitoring, ideally visualized for easy comprehension and end-to-end contract management.

Next steps

Last episode, we introduced the ink! smart contract via a Blockchain Food Order use case, with a focus on leveraging ink! Macros for code simplicity, reusability, and readability. This is part 2 of the same Learn ink! By Example series, specifically covering ink! smart contract upgradability and storage layouts, particularly in comparison to Solidity contract upgradability.

We have made this an open-source project under the Apache 2.0 license, inviting both web2 and web3 developers to learn and contribute. The third and final part of this series will focus on ink! smart contract benchmarks, measuring the performance and gas costs of various optimization implementations.

Keep an eye out for the upcoming article. Your feedback, suggestions, and contributions are welcome!