

Learn ink! by Example — Order Food on the Blockchain



Why learn ink!

There are many blockchain smart contracts, such as [Solidity](#), [Vyper](#), [Yul](#), [Cairo](#), [Move](#), and [ink!](#), etc., that support different virtual machines running on different blockchains, each with their own pros and cons,

When compared to Ethereum and its EVM, the [Polkadot](#) blockchain, with its unique architecture, [WASM](#) (Web Assembly) virtual machine, and [ink! Smart contract language](#) is a growing ecosystem that is attracting increasing developer adoption.

[Parity](#) Technologies, the company behind Polkadot and led by Dr. Gavin Woods, has the ambition to build the foundation for Web 3.0. From the Substrate blockchain framework to Polkadot to ink!, its sharded protocol and tools enable parachains to operate seamlessly at scale. In this article, [Why we believe in Wasm as the base layer of decentralized application development](#), Parity shared its belief and offered ink! as a solution.

[WebAssembly](#) (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on all devices, from servers to browsers, with great portability. It has

already been endorsed by industry heavy weights and popular in the web2 world, with increasing indications that the Wasm VM is gaining traction as an EVM alternative. For example, Abitrum adopted Wasm for fraud proof, in parallel to the same smart contract code compiled to run on EVM.

[ink!](#) Is a Rust-based domain specific language (DSL) for writing Wasm based smart contracts for Substrate blockchains. Compared to Solidity, Ink! Is a relatively new kid, yet with cool features and great potential. Being a community-focused project, it needs developers to share valuable learning experiences and input for continuous improvements. This article is meant to do exactly that, with objective learning and development experiences, evaluating and climbing the ink! learning curve.

Learning by doing in 3 stages

The project's plan is to come up with an intuitive example, implement in ink!, and share learnings with the community. It is intended to be a technical showcase with simple yet decent complexity to demonstrate three chosen aspects, as compared to Solidity as a contrast.

The three aspects to be demonstrated are:

- ink! Smart contract and macros for code simplicity, reusability and readability.
- Upgradability of Ink! based smart contracts.
- Performance benchmark and gas cost savings.

There's a lot of details to go over, so we'll progress in corresponding three stages, with takeaways documented in a series of three articles, each covering respective areas. This is the first of the series, focusing on macros in ink! smart contract development.

The series could potentially benefit several types of audiences:

- Developers wanting to check out alternative smart contract languages.
- Rust developers want to do blockchain based smart contract development.
- Ink! novice wanting to learn more about declarative and procedure macros.
- Solidity / Vyper developers compare with other smart contract development paradigms.
- Network operators wanting to know more about smart contract upgrade practices.
- Web2 developers want to get a taste of coding smart contracts using their familiarity. programming languages such as Rust.
- EVM or compatible VM developers concerning performance and gas fees.
- Blockchain researchers surveying on vs off-chain computing and storage.

The use case: Order Food on the Blockchain

Current ink! document uses a very simplistic example called Flipper, which switches a boolean on and off. It lacks the sophistication to showcase the full potential of the ink! language. In this example, we came up with a contrived workflow example, where multiple parties: restaurants, couriers, consumers and dApp admin interact to register, order, prepare and deliver food on chain.

We chose to define and implement this use case for multiple reasons:

1. A more sophisticated business workflow with multiple parties leveraging blockchain to conduct transactions
2. There are reusable business logic to be encapsulated and implemented in macros
3. Demonstrate the end-to-end process of Wasm based smart contract upgrades, and best practices to best preserve states and to avoid hard fork
4. Implementation best practices demonstrating techniques to improve performance, reduce binary footprint, and lower gas fees

The main flow of the Order Food on Blockchain example is as follows:

1. Registration:

- Restaurants register on the dApp by providing their details such as name, address, and contact information.
- Consumers register on the dApp by providing their details such as name, address, and contact information.

2. Ordering:

- Consumers browse the available restaurants and select the desired items from their menus.
- Consumers place an order by specifying the items, quantities, and delivery address.
- The order is stored on the blockchain, along with the restaurant and consumer information.

3. Preparation:

- Restaurants receive the order details and start preparing the food.
- Restaurants update the status of the order to indicate that the food is being prepared.

4. Delivery:

- Couriers are assigned to deliver the food based on their availability and proximity to the restaurant and consumer.
- Couriers update the status of the order to indicate that the food is out for delivery.
- Couriers deliver the food to the specified delivery address.
- Couriers update the status of the order to indicate that the food has been delivered.

5. Payment:

- Consumers make payment for the food using a supported cryptocurrency.

- The payment is stored on the blockchain, along with the order and consumer information.
- Restaurants receive the payment for the delivered food.

Throughout this process, the dApp admin oversees the entire workflow and ensures that all parties are following the rules and guidelines set by the smart contract.

The specific code for implementing this example can be found on GitHub, where the open-source nature of the project allows for collaboration and improvement of the codebase. This example demonstrates the capabilities of the ink! language in handling complex business workflows involving multiple parties and showcases best practices for smart contract upgrades, state preservation, and gas fee optimization. Here's the main flow of this Order Food on Blockchain example, with specific code open sourced and available in Github.

Personas

- Consumers: These are individuals who want to buy food. They can submit orders, pay for the food and delivery fees using tokens supported by the blockchain network. They can also confirm delivery.
- Restaurants: These are businesses that offer food. They can add, update, or delete items from their menu, including dish names, descriptions, prices, and estimated preparation times. They can also confirm orders, prepare food, request delivery, and get paid.
- Couriers: These are individuals who deliver food to customers. They receive requests for delivery and get paid a service fee, which is a percentage of the order total, upon successful delivery.
- dApp manager: This is the administrator of the decentralized application. They handle tasks such as access control, listing restaurants, couriers, and customers, approving service fee rates, and changing ownership of the smart contract.
- dApp users: These are individuals who use the decentralized application. They can self-register with a wallet and provide their information. They can view all restaurants, food items, and couriers. They can also register to be service providers (restaurants or couriers) or submit orders as customers.

Workflow

- The example implements a mini workflow for a marketplace with service providers and consumers interacting with each other on the blockchain.
- Service providers can register or update their services, such as restaurant information, menus, and delivery rates.
- Consumers can browse available restaurants and food, place orders, and select a courier for delivery.

- All participants in the marketplace have registered wallet addresses for sending and receiving payments.
- The decentralized application provides escrow functionality, holding the payment for an order until the food is picked up and delivered. The proceeds are then split, with the food subtotal going to the restaurant and the delivery charges going to the courier.
- The mini workflow is implemented as a state machine in the smart contract storage. Each state transition, such as order submission, confirmation, food preparation, delivery, and acceptance, is captured on-chain. Events are emitted by the smart contract for each state transition or error, triggering the next step in the state machine.
- This example is meant for learning and sharing, with a moderate level of complexity to demonstrate the implementation using ink! (a Rust-based smart contract language). In the real world, blockchain can be used for trustless transactions and enterprise-grade workflows.
- The open source community can further develop this example by adopting a workflow specification written by non-technical users or generated through a workflow DSL (domain-specific language) to smart contract compiler. This would allow businesses to conduct transactions on the blockchain using an intuitive frontend and tools. Enterprises of all sizes can leverage the open ledger as a trust layer, interacting and transacting transparently for increased efficiency, scalability, and cost-effectiveness.

ink! Implementation

The example provided in this text is implemented using the ink! smart contract language, which is built on top of the Rust programming language. Developers familiar with Rust will find it easy to read the open source code as ink! is a Rust-based DSL (Domain Specific Language).

For non-Rust based smart contract developers, it can still be a valuable exercise to experiment with the code and sandbox to get a sense of how ink! based smart contract development compared to other languages like Solidity. More information about the performance benchmark, binary size, and gas costs of this smart contract will be shared in part three of this series, along with various optimization techniques.

Here are some highlights of the data, services, security, and access control implemented in this ink! smart contract:

- Top-level data types, such as Food, Order, Delivery, Customer, Restaurant, Courier, and Manager, are implemented using openBrush's Mapping feature.
- The smart contract defines and implements multiple service categories, including Restaurant services, Courier service, Customer services, Payment services, and Manager services.
- Data is persisted in the smart contract's storage, while services are transactions that are added to blocks.
- The smart contract uses openBrush's ownable feature to set and get owners with corresponding access control to create, read, update, and delete data. For example, only

restaurant and courier owners can administer their respective services.

The entire [codebase](#) for this example is available on GitHub under the Apache 2 license.

Security & Access Control

[OpenBrush](#) is a smart contract development framework for ink! that provides a number of features to help developers build secure and reliable contracts. It is [an open source project](#) aimed to provide similar capabilities as [OpenZeppelin](#) for Solidity smart contract developers.

OpenBrush attempts to provide similar security and access control features like OpenZeppelin for ink! smart contract development: [Ownable](#) extensions and [AccessControl](#) with [AccessControlEnumerable](#) extension,

Ownable

Ownable is for implementing ownership in smart contracts. This is the most common and basic form of access control, where there's an account that is the owner of a contract and can administer it. OpenBrush's Ownable contract provides the following security features:

- A public modifier called `onlyOwner` that can be used to restrict access to certain functions.
- A public function called `transferOwnership` that can be used to transfer ownership of the contract to another address.
- A public function called `isOwner` that can be used to check whether the current account is the owner of the contract.
- A public function called `renounceOwnership` that can be used to renounce ownership of the contract.

The `renounceOwnership` function is a unique feature of OpenBrush's Ownable contract. This function allows the owner of the contract to permanently give up ownership of the contract. This can be useful in situations where the owner no longer wants to be responsible for the contract.

Access Control

Role-Based Access Control (RBAC) offers flexibility in defining multiple roles. OpenZeppelin's `AccessControl` library provides a number of features to help developers control access to their contracts.

- **Roles:** Developers can define roles for different users and groups of users.
- **Permissions:** Developers can assign permissions to roles, such as the ability to transfer tokens, create new contracts, or call specific functions.
- **Policies:** Developers can define policies that control how permissions are applied. For example, a policy might require that two approvers sign a transaction before it can be executed.

OpenBrush aims to provide a comprehensive set of security and access control features for ink! smart contract development, similar to what OpenZeppelin offers for Solidity smart contracts. It includes the Ownable contract for ownership management and the AccessControl library for role-based access control. Additionally, OpenBrush extends these features with unique functionalities such as the ability to renounce ownership, timed permissions, multi-factor authentication, and audit trails.

- Timed permissions: Developers can specify that permissions expire after a certain amount of time.
- Multi-factor authentication: Developers can require users to provide multiple forms of authentication, such as a password and a one-time code, before they can access a contract.
- Audit trails: Developers can track who has accessed their contracts and what they have done.

ink! macros

ink! macros provide a powerful tool for developers to encapsulate reusable logic and improve the simplicity, reusability, and readability of smart contract code. They can be used to define custom syntax or code that can be reused within a smart contract, and they are translated into actual Rust code during compilation.

One of the advantages of ink! macros is their ability to handle a variable number of parameters, which allows for flexibility in processing different items or operations. This can be especially useful for tasks like printing or iterating through a set of items.

The reusability of ink! macros is important for adoption, as it allows developers to modularize related functionalities and share them across different projects. This can accelerate smart contract development by providing easy access to mature code snippets.

Using ink! macros can also lead to better code quality and lower gas fees. By encapsulating logic in macros, developers can reduce repetition and inconsistency in their code, leading to cleaner and more maintainable contracts. Additionally, optimizing the implementation of macros can help reduce gas fees.

Another benefit of ink! macros are their ability to separate the layers of smart contract development and abstract business logic in dApps. This can make it easier to divide and conquer complexities and facilitate functional invocation among smart contracts, even across different chains.

In this example, to add another menu operation, for example, adding food pictures for each menu item, a developer can expand the same macro to support additional functionalities, making it easier to share among dApp developers to invoke the macro with increasing operations as a reusable component similar to code library. This is why this exercise comes up

with specific macro implementations to illustrate its potentials not just at system level, but also encapsulating business logic in dApps, with multiple benefits:

- ink! macros, similar to “libraries”, can help to improve smart contract readability, reusability and maintainability reducing repetition and inconsistency
- Optimally implemented macros can reduce gas fees and improve code quality
- Lower ink! learning barrier and increase community adoption
- Flexibility to handle variable number of parameters, which functions can't
- Extensibility to incrementally add more functionalities to the macro
- Separation of smart contract dev layers to divide and conquer complexities
- Abstraction of business logic in dApps to facilitate functional invocation among smart contracts, even cross chains, following web3 design patterns.

Payment_macro

This is a declarative macro to abstract payment functions. The code use to call separate functions to pay restaurants and couriers respectively, both now being consolidated into the same `transfer_from_contract_to_account` macro defined in [helpers.rs](#):

```
# [macro_export]
macro_rules! transfer_from_contract_to_account {
    ($account:expr, $amount:expr) => {
        if T::env().transfer($account, $amount).is_err() {
            return core::prelude::v1::Err(FoodOrderError::NotTransferred);
        }
    };
}
```

crud_macro

This explains the concept of a procedure macro and its use in abstracting multiple business functions into a single macro. Unlike declarative macros that match expressions with predefined patterns, procedure macros manipulate the Abstract Syntax Tree (AST) of the compiler to rewrite code. The code is auto-generated during compilation time.

The example provided demonstrates how a procedure macro can encapsulate business logic to simplify smart contract code at the application level. Previously, there were separate functions for creating, reading, updating, and deleting food menu items in a restaurant. These functions are now folded under a single macro called "ink!" which can invoke any Create, Read, Update, or Delete function. This macro provides a level of abstraction to encapsulate food operations.

The procedure macro is defined in the "[crud-macro/src/lib.rs](#)" file. An example of the "update_food" macro is shown, which includes specific data quality checks, identifier matches,

and error handling. In the "[restaurant_service.rs](#)" file, the macro is invoked without dealing with low-level details, improving code simplicity and readability.

```
#[proc_macro_attribute]
pub fn update_food(_attr: TokenStream, item: TokenStream) -> TokenStream {
    let input = parse_macro_input!(item as ItemFn);
```

```
#[update_food]
#[modifiers(is_restaurant_user)]
fn update_food(
    &mut self,
    food_id: FoodId,
    food_name: String,
    description: String,
    price: Balance,
    eta: u64,
) -> FoodResult {
    // Emit `UpdateFoodEvent`
    self.emit_update_food_event(food_id, food_name.clone(),
                                description.clone(), price, eta);
}
```

The same macro also includes "create_food" to insert food items into the restaurant menu and "delete_food" to remove deprecated menu items. More functions can be added incrementally as needed. If there are multiple invocations, the developer only needs to modify the macro code instead of changing scattered function calls, reducing the chances of errors and making maintenance easier.

Testing

This project leverages [Chai](#) as the testing framework for end-to-end testing with [testing scripts, assets and artifacts](#) in the Github repo. A testing script on a happy path shows the following output:

```
foodorder test
```

Main Functionality

- ✓ Platform is ready
- ✓ Restaurant A is added (...ms)
- ✓ Courier A is added (...ms)
- ✓ Customer A is added (...ms)
- ✓ Food A is added (...ms)
- ✓ Order is submitted (...ms)
- ✓ Order is Confirmed (...ms)
- ✓ Food is cooked and Payment is transferred to restaurant (...ms)
- ✓ Order is Delivered (...ms)
- ✓ Delivery is accepted and Payment is sent to courier (...ms)

```
10 passing (..s)
```

Deployment

[Shiden](#) Network is a multi-chain decentralized application layer on Kusama Network. [Shibuya](#) is the Shiden's parachain testnet with EVM functionalities. We choose it for deployment as Shiden supports EVM, Wasm, and Layer2 solutions. We prepared a [shared document with screenshots](#) to illustrate the flow of deploying the contract then step-by-step flow of ordering food on the blockchain. Here's the expected output when running test from the docker image:

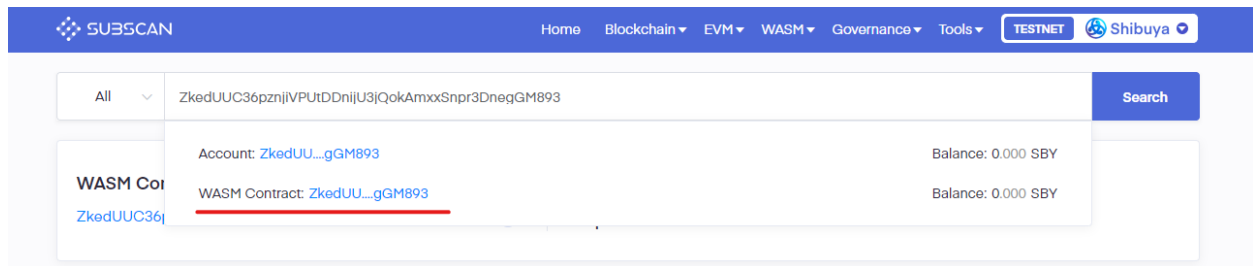
- ✓ Initialising OK
- ✓ Getting WASM OK
- ⚡ Connecting to node2023-08-11 11:47:18 API/INIT: shibuya/105: Not decorating runtime apis without matching versions: EthereumRuntimeRPCApi/5 (4 known)

- ✓ Connecting to node OK
- ✓ Deploying OK
- ✓ Writing config OK

Contract deployed!



Contract address: Yn1dHJTbKuMhA6rLLsRXQtDu4mSFGC6xtvDTueNz1axJ5Dz

After successfully deployed, you can check the deployed contract on the shibuya blockexplorer <https://shibuya.subscan.io/>.



Once a contract is deployed, either one you just deployed at a new address, or an [existing Wasm contract like this one already deployed](#), at address Yn1dHJTbKuMhA6rLLsRXQtDu4mSFGC6xtvDTueNz1axJ5Dz. It looks like the following with 31 messages listed under the FOODORDER Wasm contract.

contracts

 CONTRACT1	Messages (31) ▼
 FOODORDER	Messages (31) ▲
▶ exec	<code>courierService::pickupDelivery</code> (deliveryId: <code>PickupDeliveryInput1</code>): <code>Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>
▶ exec	<code>customerService::acceptDelivery</code> (deliveryId: <code>AcceptDeliveryInput1</code>): <code>Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>
▶ exec	<code>customerService::addCustomer</code> (customerName: <code>AddCustomerInput1</code> , customerAddress: <code>AddCustomerInput2</code> , phoneNumber: <code>AddCustomerInput3</code>): <code>Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>
▶ exec	<code>customerService::submitOrder</code> (foodId: <code>SubmitOrderInput1</code> , deliveryAddress: <code>SubmitOrderInput2</code>): <code>Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>
▶ exec	<code>managerService::addCourier</code> (courierAccount: <code>AddCourierInput1</code> , courierName: <code>AddCourierInput2</code> , courierAddress: <code>AddCourierInput3</code> , phoneNumber: <code>AddCourierInput4</code>): <code>Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>
▶ exec	<code>managerService::addRestaurant</code> (restaurantAccount: <code>AddRestaurantInput1</code> , restaurantName: <code>AddRestaurantInput2</code> , restaurantAddress: <code>AddRestaurantInput3</code> , phoneNumber: <code>AddRestaurantInput4</code>): <code>Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>
▶ exec	<code>managerService::changeFeeRate</code> (rate: <code>ChangeFeeRateInput1</code>): <code>Result<Result<Text, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>
▶ exec	<code>managerService::changeManager</code> (newAccount: <code>ChangeManagerInput1</code>): <code>Result<Result<Text, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>
▶ exec	<code>restaurantService::addFood</code> (foodName: <code>AddFoodInput1</code> , description: <code>AddFoodInput2</code> , price: <code>AddFoodInput3</code> , eta: <code>AddFoodInput4</code>): <code>Result<Result<u64, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError></code> ⓘ <i>No documentation provided</i>

To start interacting with the system, you can manually read or execute messages with parameters using different accounts that represent various roles such as restaurants, couriers, and consumers. I recommend installing the [Sub Wallet](#) as a browser extension and obtaining some free SBY tokens from a [faucet](#) since we deployed on the Shibuya testnet, which uses SBY tokens.

Once you have the necessary setup, you can run the example's order food on the blockchain workflow by invoking any of the 31 messages that play different roles. Please note that reading messages does not require gas fees, but executing messages does. We have prepared a [shared document with screenshots](#) to guide you through the process of deploying the contract and step-by-step instructions for ordering food on the blockchain. Towards the end of the workflow, you will see the order's status advancing to "FoodDelivered," as shown in the screenshot below.

call a contract

contract to use

FOODORDER

amcaMHvzD...

call from account

ACCOUNT 1 (SUBWALLET-JS)

transferrable 15.3220 sBY

ZWexd44p3r...

message to send

getService::getOrderAll (from: GetOrderAllInput1, to: GetOrderAllInput2):

Result<Result<Vec<LogicImplsTypesOrder>, LogicImplsTypesFoodOrderError>, InkPrimitivesLangError>

from: GetOrderAllInput1

0

to: GetOrderAllInput2

10

max reftime allowed (m)

1

max read gas

☒

max proofsize allowed

1000000

0.400s execution time, 9.99% of block weight

Call results

getService::getorderall (from: 0, to: 10): result<result<vec<logicimplstypesorder>, logicimplstypesfoodordererror>, inkprimitiveslangerror>

{Ok: {Ok: [{foodId: 1 restaurantId: 1 customerId: 1 courierId: 1 deliveryAddress: asfasdfsadf status: FoodDelivered timestamp: 1,691,783,334,608 price: 1,000 eta: 100 }]}}

8/12/2023 4:23:21 am

Read

Recommendations and Next Steps

This project aims to provide a comprehensive learning experience for the ink! language and related tools by implementing an Order Food on Blockchain example. The project will focus on three specific areas: ink! macros, Wasm smart contract upgradability, and binary footprint/gas costs/performance benchmarks.

The Order Food on Blockchain example will showcase the capabilities of ink! smart contracts by allowing users to place food orders on a decentralized platform. The ink! macros will be used to simplify the development process and enhance the readability of the code.

The project will also explore the concept of Wasm smart contract upgradability in part two, which allows for the seamless upgrading of smart contracts without disrupting the existing functionalities. This feature will be implemented to ensure the scalability and adaptability of the

Order Food on Blockchain platform.

Additionally in part three, the project will analyze the binary footprint, gas costs, and performance benchmarks of the ink! smart contracts. This analysis will provide insights into the efficiency and cost-effectiveness of the platform, ensuring optimal utilization of blockchain resources.

This project will serve as a learning opportunity for individuals interested in understanding ink! language, ink! macros, Wasm smart contract upgradability, and optimizing binary footprint, gas costs, and performance benchmarks in the context of a real-world use case.

ink! learning resources

To get started with understanding ink!, a good starting point is the "[Guided Tutorial for Beginners](#)" and the post "[What is Parity's ink!?](#)" by Michi Müller. These resources provide a high-level understanding of ink!.

For advanced Rust developers, there is a GitHub markdown that covers ink! architecture and internals, which can be helpful for diving deeper into ink!.

For ink! 4.0 specifically, there is an [announcement](#) that outlines what's included in the release.

Parity Technologies has ink! documentation in their [GitHub repository](#), which covers ink! basics and provides some examples.

Substrate's documentation site also has [a smart contract tutorial](#) that covers specific topics on how to use a smart contract template, store and retrieve values, and add functions to a smart contract.

If you're looking for ink! projects, there is a [curated list](#) on GitHub that includes production and testnets, dApps, libraries and standards, DeFi, gaming, and block explorers.

For tools, OpenBrush has documented its [ink! smart contract libraries](#), including [ownable](#) and [access control](#). [Contracts UI](#) provides a frontend for contract deployment and interaction.

If you have specific questions or need help, you can visit the [Stack Exchange for Substrate](#) and Polkadot, where there are over 400 questions tagged with ink!.

While [ink!'s official documentation](#) is a great source to learn ink! as a smart contract language, it may lack sufficient coverage of macros within the context of ink!. However, there are resources available on Rust macros that can be helpful. The Rust Programming Language has a [document](#) that explains what macros are and the difference between macros and functions. The [Little Book of Rust Macros](#) is another excellent source specifically focused on Rust macros,

although it may not cover macros in the context of ink! smart contract language. These resources should provide a good starting point for learning and understanding ink! and its various aspects.

Recommendations

ink! is seen as a promising smart contract language that provides an alternative option for developers familiar with Rust. However, there are several areas where improvements can be made to enhance the ink! ecosystem and attract more developers:

1. **Learning:** The learning curve for ink! and its associated tools can be steep, especially for developers not familiar with Rust. Having a well-organized learning center with in-depth resources and examples would greatly benefit the community and make the onboarding process easier.
2. **Tooling:** While there are some tools available for ink!, there is room for improvement in areas such as upgrade management, interoperability, gas estimation, performance benchmarks, and industry-specific smart contract macros and templates. Enhancing the tooling will make it easier for developers to adopt ink! for building dApps.
3. **Ecosystem:** The Solidity ecosystem is more mature compared to ink!, with a wider range of libraries, frameworks, tools, and job opportunities. Building a robust ecosystem around ink! would make it more attractive to developers as a viable option for smart contract development.
4. **Enhancements:** There are other smart contract languages and frameworks, such as Move and Arbitrum Nitro stack, that offer innovative features like security, auditability, and resource-oriented design. ink! can draw inspiration from these innovations beyond just the Substrate framework.
5. **Adoptions:** While ink! has been well received by a niche group of Rust developers implementing smart contracts on Polkadot/Kusama, expanding the community requires reaching out to target segments beyond this small group. More efforts are needed to win incremental adoptions and grow the ink! community.

To address some challenges that may hinder adoption, the ink! project should focus on:

- **Community:** The [open-source project](#) is currently concentrated within Parity Technology and a few Dotsama parachain development shops. Expanding the community beyond this narrow group is crucial for growth.
- **Communication:** Clear and effective communication is essential to convince developers to adopt ink!. Issues like the [Polkadot API failing to connect to the Substrate Contracts Node docker container](#) should be documented with solutions to ensure a high level of quality and

support.

- Support: Bugs affecting important features should be clearly communicated to the community along with any workarounds and expected fixes. Lack of clarity on technical solutions can discourage adoption.

- Branding: ink! and its related tech stack may not have enough visibility outside of the Dotsama community. The branding should be more distinct to avoid confusion with unrelated search results and increase awareness of ink! and its associated resources.

- Developer experiences: The ink! community is still smaller compared to more mature smart contract languages like Solidity. Efforts should be made to catch up and provide a comparable developer experience with more content, discussion threads, tutorials, code repositories, and tools.

Despite these challenges, ink! is recognized for its technical merits and comparative advantages in smart contract development. With support from the community and continuous improvements, ink! has the potential to become a powerful tool in the web3 ecosystem.

Next steps

The project has been made [open source](#) under the Apache 2.0 license, allowing both web2 and web3 developer communities to access and contribute to it. This article will be split into three parts. The first part covers the use case and the implementation of ink! with macros. Upcoming part 2 will focus on upgradability, and part 3 will benchmark the performance, measure binary footprints, and optimize gas costs in different ink! optimization variations.

Keep an eye out for the upcoming articles and feel free to provide feedback, suggestions, and contributions to the community!

Please note that this document is a draft for an article that will be published on [Medium](#).