DevOps for Blockchain Smart Contracts

Ben Burns, Head of Blockchain Development, Truffle
David Burela, Sr. Software Engineer, CSE, Microsoft
Marc Mercuri, Principal Program Manager, Blockchain Engineering, Microsoft

Introduction

DevOps is the union of people, process, and products to enable continuous delivery of value to end users. Many organizations have already moved to embrace a DevOps culture, and activities such as continuous integration and continuous delivery are now firmly entrenched in how they develop software.

Blockchain has emerged from the shadow of its cryptocurrency origins to be seen as a transformative data technology that can power the next generation of software for multi-party Enterprise and consumer scenarios. With the trust and transparency that blockchain can deliver, this shared data technology is seen as a disruptor that can radically transform assumptions, costs, and approaches about how business is done.

With the mainstreaming of blockchain technology in Enterprise software development, organizations are asking for guidance on how to deliver DevOps for blockchain projects and products.

This paper looks at core aspects of DevOps with a focus on considerations and approaches to incorporate continuous delivery, continuous improvement, and infrastructure as code to the development of smart contracts for blockchain solutions.

This paper is complemented by an implementation guide that showcases how to deliver the approaches defined in this document with a mix of Azure DevOps, VS Code and popular open source tools such as the Truffle Suite.

The Value of Blockchain

For those new to blockchain, it may be helpful to first look at its application in a real world scenario, such as the buying and selling of shares of stock, to understand where and how the technology delivers value.

Sally would like to sell 100 shares of Microsoft stock and John would like to buy 100 shares of Microsoft stock. John and Sally don't know each other so they turn to brokers to help them discover reputable buyers and sellers. John works with Woodgrove Bank and Sally works with Contoso Investments to facilitate a trade. The trade appears immediate, but the reality is that it may not "settle" for 3 days. This is not ideal for either party – Sally doesn't have access to her funds and John can trade that stock for 3 days.

The reason for the extended settlement time is that the brokers may not know or trust each other, and to settle transactions they work with a third party clearinghouse to settle those trades.

Beyond the issue of time for the buyer and seller, the brokers and the clearinghouse are not using a single source of truth – they all build their own systems with their own databases. A common occurrence in this scenario is that databases between organizations get our of synch and introduce

overhead costs for reconciliation amongst the parties. It can also introduce delays in dispute resolution if parties have a different view of the "truth" due to disparities across the databases. Some financial firms have over 1000 people tasked solely with addressing issues tied to this type of reconciliation.

Where Blockchain Can Help

Blockchain delivers a shared ledger between the participating parties, providing transparency and consistency of data. It is trusted as a single source of truth by participants because the data is both immutable and cryptographically secure.

With trust and transparency, the incorporation of blockchain into the solution can reduce settlement times and dramatically reduce costs[1] tied to reconciliation. If there are attestable, shared sources of truth that identify that Sally owns the stock and John has the money to buy the stock, there is the potential to also limit the need for brokers in many scenarios which can further expedite settlement times and reduce costs.

DevOps for Smart Contracts

The core concepts and activities to deliver DevOps for blockchain solutions are no different than they are for other projects.

Developers have a local copy of code, make changes to that code, run local tests, and upon successful completion of those tests they submit a pull request.

That request to merge their code into a shared code repository can go through a set of gated approvals and through a set of automated tests in one or more build pipelines.

Based on the success of the build pipeline(s), release pipeline(s) would then deliver the code to one or more environments. Just as in non-blockchain projects, the DevOps approach to build out these environments with an "infrastructure as code" approach. This uses automated deployment of environments to deliver components, topology and configuration which are consistent, and avoid potential "drift" in these areas that can occur by manual changes to an environment.

While the core concepts are the same, the multi-party nature of blockchain applications and the immutability of the ledger do present a set of blockchain specific considerations that should be considered when determining the DevOps approach for blockchain solution.

Terminology Considerations

At its most basic, developing solutions for a consortium of participants that will run on an immutable ledger benefits from upfront agreement on terminology.

Imagine, for an example, where a group of companies establish a blockchain network focused on supply chain scenarios. The associated smart contracts will have functions and properties that implement the logic and state.

Contoso, Woodgrove, and Tailspin are three members of that consortium and they all sell the same widget. While the widget is the same, the way these organizations describe it internally is different –

[1] Because not all data, such as Personally Identifiable Information (PII), can go on the ledger there will likely still be some reconciliation for this type of data that will continue to be held "off chain", e.g. in a relational database.

Contoso calls it an item, Woodgrove calls it a product, and Tailspin calls it a SKU. Gaining agreement amongst members on a set of consistent terminology to be used in the smart contracts can help reduce potential disconnects and time-wasting disputes later in the development process.

Roles, Responsibilities, and Policies

In many consortiums, members do not have the same role, level of ownership, or responsibility for DevOps related activities.

Reaching agreement on answers to key questions will help inform a successful DevOps implementation.

Examples include –

- Are rights assigned to individuals, systems/services and/or the organization?
- What is the process for removing rights from an existing consortium member?
- What is the process for adding rights to a new or proposed consortium member?
- How will access and rights be administered at the different levels (platform, infrastructure, ledger, smart contract) of the solution?
- Who can make pull requests?
- Who can approve pull requests and what is the process by which they're approved?
    - How many members must approve a request?
    - Do different members votes have different weights?
    - Must approval be manual or can it be automated, e.g. proceed if all tests are completed successfully.
- Given consortiums often span time zones, what are the consortium's requirements for the timing of deployments?
- When issues occur, how is the above process impacted (approach, SLA, policies, etc.) for identifying, prioritizing, and delivering hot fixes to an environment?

Test Data

Another consideration that should be reviewed and documented is what data the consortium will use for development and testing.

Populating test data for a blockchain is more complex than other types of data stores. In a relational database, for example, a script could create database tables and pre-populate them with INSERT statements written in SQL.

Populating blockchain with scripts, however, is more complicated. The desired base state for a populated blockchain typically represents a set of signed transactions from multiple parties. These take place in the context of a business process that has is represented in one or more smart contracts. Transactions will occur in the context of blocks, so populating a blockchain in a similar fashion could be done but require coordination and sequencing of transactions and done across the context of multiple parties.

An emerging trend to achieve the same result is to deliver a snapshot (referred to as a "fork") of an existing blockchain. This can often deliver the same result without the complexity of building a system to create synthetic transactions to populate the ledger.

In some cases, the fork will be of the public chain or a production blockchain of a consortium. For some Enterprises, there are constraints on production data being used for testing purposes. In these cases, the fork is of a blockchain setup for use as a "test net."

Defining a Key and Secret Management Strategy

While not all blockchain solutions will expose keys to the consumers of their service, keys will be present in every scenario. Depending on the specifics of the end to end solution, there will also likely be certificates, cryptographic and other secrets, and API keys which need to be managed securely.

Key areas of consideration and associated actions for a strategy should include -

- **Secrets Management** – Determining approach to securely store and tightly control access to tokens, passwords, certificates, API keys, and other secrets used in the solution.
- **Key Management** – Agreeing on approach to create and control the encryption keys used to encrypt data or sign transactions.
- **Certificate Management** – Determining approach (or vendor) to provision, manage, and deploy public and private Secure Sockets Layer/Transport Layer Security (SSL/TLS) certificates for use with the consortium's cloud providers and internal connected resources.
- **Determine if Hardware Security Modules are required**- Determining if the secrets and keys must be protected either by software or FIPS 140-2 Level 2 validated HSMs.

Testing Shared Assets

It's also important to establish agreement on what to test and how to test it. Different organizations may have different priorities and it is important to establish a common understanding for what should be tested for shared assets like smart contracts.

Testing Consortium Specific Assets that have Dependencies on Shared Assets

While smart contracts represent shared state and logic for business scenarios between participants in a blockchain network, it is not uncommon for consortium members to interact with those smart contracts via existing applications specific to their organization.

Beyond the development, operations, and support costs for developing a new application, another key driver here is avoiding having to retrain staff.

This impacts testing as the DevOps strategy should also include a focus on where to trigger the testing of the consortium specific code (applications, services, code) that could be impacted by changes to shared assets.

Release Definition and Delivery

Consortiums can have dozens of members with competing priorities and different views on scope and priority of releases.

It is important to document agreements on how the multiple organizations in the consortium define and reach agreement on scoping and prioritizing features, scoping releases, the process for releasing software for different environments and any special considerations for the timing of deployments to shared environments for test and production.

Smart Contract Design

Blockchain applications are often designed to handle financial transactions, track mission-critical business processes, and maintain the confidentiality of their consortium members and the customers they serve. Software faults in these areas might, in extreme cases, represent an existential risk to your company or your consortium. As a result, blockchain applications usually demand a more rigorous risk management and testing strategies than traditional software applications.

One of the largest drivers of risk in blockchain software is complexity. Complexity in software is like entropy in the physical world. Energy must be expended if you wish to reduce it. At every change to the system, from requirements discovery, to implementation, to final test and auditing, and even post deployment, complexity will increase if pragmatic steps aren't taken to manage it.

Some complexity reduction strategies you might employ are:

- Track requirements from day one, updating them as new requirements are discovered.
- Minimize the set of concerns which a single contract must handle, preferring many simpler smart contracts to a few larger ones.
- Keep complex aggregate or inferred state calculations off chain, pushing them on chain with trusted oracles where necessary.
- Prefer audited and production-tested library contracts for common concerns such as access control and standardized contract implementation (e.g. ERC-20).
- Write contracts such that they can be tested with few, or no dependencies.
- Use peer reviews during development with a critical focus on reduction of unnecessary complexity.
- Consider bringing in one or more smart contract auditors, ideally early on in the development process.

It's important to remember that "less complex" doesn't strictly imply "less code." Good software will be factored out into components which are each well specified and easy to comprehend. It's quite often the case that this will result in more code overall as compared to a system which is designed to achieve the same goals but which is written in such a way as to minimize the number of components.

Design for Composability

A smart contract often defines a representation of a business process and aspects of that business process could have variants based on geographies, regulations, or other conditions that would alter the process in a specific context.

Contracts as Microservices

A popular approach is to look at smart contract design with a similar approach to how you would look at microservice design. Decompose the solution into its core entities and the processes which act on those entities, then develop discreet, composable smart contracts for each entity and process so that they can evolve independently over time.

Examples include-

- A retailer would like to sell digital copies of films to its customers. A movie studio provides blocks of licenses for this purpose but based on the business relationship between the studio and the retailer, different processes may be followed. For example, some retailers must pre-pay for new blocks of licenses while other retailers have a post-pay agreement where they can get licenses "on demand" and sent invoices for payment. In this case, the business process can be de-composed into a licensing smart contract which is complemented by pre-pay and post-pay processes broken out as companion smart contracts.
- In a supply chain, there are models where commodities are sold directly from an individual farmer, from a co-operative collection of farmers, or from corporate mega-farms. The way each of these should be handled could be different and can vary over time. While much of the end to end supply chain may be consistent, this "first mile" of the process focused on acquisition can be handled differently and having variants of contracts for each of these scenarios is appropriate.
- In many scenarios, such as those that involve financial transactions, capabilities delivered by smart contracts may need to be implemented differently for different geographies. This can be the result of the maturity of market and associated processes or the influence of regulations on the consortium members who do business in that geography. If a significant amount of the smart contract is the same across geographies, breaking these geo-specific variances out into their own smart contracts will often make sense. A core smart contract can be developed and companion smart contracts could be created for sections where there are geographic variances, e.g. a payments contract and then companion contracts for the United States, China, and Russia.

When defining smart contracts, decomposition should be a key consideration and the extent of decomposition should be driven by a conscious trade off of granularity vs. throughput that's acceptable to your project requirements.

Designing for Upgradability

In non-blockchain solutions, delivery pipelines are used to deliver updated code and configuration to a hosting environment, e.g. virtual machine (vm), container, or serverless offerings such as Azure Functions. For changes or addition to data technologies, new instances of data stores will be created with updated logic and any relevant data will be migrated using scripts or tooling.

Blockchains are distributed ledgers whose history immutable – once something is written to the chain there is a permanent recording of it. This includes smart contracts.

Over time, smart contracts will need to evolve due to requirements driven by the consortium or regulatory bodies that govern members. As smart contracts represent business processes that may span days, weeks, months or even years, there may be cases where pre-existing contracts are allowed to continue in their current form and all new instances will use a new version.

In other cases, there will need to be a migration of state from existing versions of the smart contract to the new version, and the original contract will need to have properties that both identify the change and help dispatch requests to the new instances of the contract.

Implement RBAC Where Appropriate

Role Based Access Control (RBAC) in smart contracts is used to help define which role(s) can call specific functions. This enforces the rules of the business process and ensures bad actors don't compromise the integrity of a contract.

Tests can then be defined that validate not just that the functions return the expected result, but also test that only appropriate parties can call them.

Depending on the targeted blockchain stack for your solution, there may be smart contracts already available in open source community that address RBAC. For example, if writing smart contracts in Solidity targeted for Ethereum, the OpenZeppelin project in Github provides a library for defining roles as well as assigning ownership.

Automated Testing

Paramount to any good risk management strategy is automated software testing. A good automated testing strategy aims to achieve the following **aspirational** goals:

- Tests should be written with the assumption that someone other than the author will be fixing the fault they expose, and that this person will have very little time to do so.
- Tests should execute quickly to give developers fast feedback when errors are introduced.
- Tests should define a specification for the software they exercise.
- Test code should be highly readable.
- A failing test should tell you not only that there is a fault, but also exactly where the fault lies.
- When something breaks, only one test case should fail.
- It should be possible to execute any one test case in isolation without relying on state set up by other test cases.
- Every branch in the contract logic should be covered.
- Assertions should be made which ensure that things which should happen do happen, and the things which aren't expected to happen don't happen.

Tooling and SaaS

An implementation of a DevOps strategy should include local development and testing, testing at the consortium level, and testing by consortium members for member-specific applications.

The approach reflected in this whitepaper and the accompanying implementation guide focuses on development using Visual Studio Code, Truffle Suite, and Azure DevOps to deliver the necessary capabilities.

Visual Studio Code

Visual Studio Code is a free download that provides an integrated development environment on Windows, Mac, and Linux.

Visual Studio Code provides out of the box support for Git. It also has a wide assortment of extensions that can be downloaded for additional language support, extended integration with your source code repository, in tool management of pull requests, monitoring of builds, etc.

For smart contract development specifically, VS Code has extensions for many target smart contract languages, including -

- Solidity for Ethereum
- Java and Kotlin for Corda
- Go for Hyperledger Fabric

Truffle Suite

Truffle Suite is a very popular open source suite that consists of three products – Truffle, Ganache, and Drizzle. The initial version of this document focuses on the use of Truffle and Ganache specifically.

Truffle

Truffle is a development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make life as a developer easier.

Truffle capabilities include -

- Built-in smart contract compilation, linking, deployment and binary management.
- Automated contract testing for rapid development.
- Debugging of smart contracts that can span multiple connected contracts.
- Scriptable, extensible deployment & migrations framework.
- Network management for deploying to any number of public & private networks.
- Package management with EthPM & NPM, using the ERC190 standard.
- Interactive console for direct contract communication.
- Configurable build pipeline with support for tight integration.
- External script runner that executes scripts within a Truffle environment.

Ganache

Ganache is a personal blockchain for Ethereum development you can use to deploy contracts, develop your applications, and run tests.

It is available as both a desktop application as well as a command-line tool (formerly known as the TestRPC). Ganache is available for Windows, Mac, and Linux. Ganache is also being distributed as an embeddable lib for use in tools & applications which would benefit from an internal single-node blockchain. The accompanying implementation guide also includes deployment of Ganache within an Azure function.

The ability to deploy locally or within a function reduces costs and management overhead typically associated with running a full blockchain node configuration. At the same time, it supports forking of an existing chain which provides the ability to do testing against a blockchain that mirrors the target environment, e.g. consortium production.

Azure DevOps

Azure DevOps Services is a cloud service for collaborating on code development. It provides an integrated set of features that are accessible through a web browser or IDE client, including the following:

- Git repositories for source control of your code
- Ability to do gated check ins which can accommodate consortium policies
- Build and release services to support continuous integration and delivery of consortium and consortium member apps
- Agile tools to support planning and tracking your work, code defects, and issues using Kanban and Scrum methods
- A variety of tools to test your apps, including manual/exploratory testing, load testing, and continuous testing
- Highly customizable dashboards for sharing progress and trends
- Built-in wiki for sharing information with your team

In addition, the Azure DevOps ecosystem provides support for adding extensions, integrating with other popular services, such as: Campfire, Slack, Trello, UserVoice, and more, and developing your own custom extensions.

The approach reflected in this whitepaper will use Azure DevOps to continuously build, test, and deploy the smart contracts. Azure DevOps also allows the definition of security policies for role based access control and gated check ins that are important in a consortium environment.

Defining Environments, Pipelines, and Policies

Definitions of environment types, their purpose and their specific attributes are also important to document.

As with all projects, there is a determination that needs to be made on how long-lived an environment will be (ephemeral or permanent). This blockchain infrastructure can be a newly deployed blockchain that has no data, a “forked” chain that effectively delivers a snapshot of a blockchain, or a full environment that includes the full set of transactions for a specific blockchain network. Whether empty or pre-populated, there should be no dispute that the environment should represent the current or planned environment for production.

The table below compares the attributes of the different types of environments.

Environment	Attributes
Local development environment	• Purpose is for dev/test by an individual developer working on a feature or fix • Infrastructure is ephemeral • Test and deployment are done against an empty, synthetic, or “forked chain” that represents production. • Build pipeline(s) include consortium and developer defined tests • Deployment pipeline(s) driven by approved and proposed policies for the consortium and those of the consortium member the developer(s) work for.

Consortium member test environment	• Purpose is test for team working on internal applications that use the smart contract(s) *at a specific consortium member* • Infrastructure is ephemeral or permanent • Test and deployment are done against an empty, synthetic, or "forked chain" that represents production. • Build pipeline(s) include includes consortium and consortium member defined tests • Deployment pipeline(s) driven by approved consortium and consortium member policies
Consortium test environment	• Purpose is to serve as a test environment *for the consortium* • Infrastructure is ephemeral or permanent • Test and deployment are done against a "forked chain" that represents production. • Build pipeline(s) include consortium approved tests • Deployment pipeline(s) driven by approved consortium policies
Consortium production environment	• Purpose is the production environment for consortium on a blockchain network used by the consortium. • Infrastructure is permanent • Deployment is done against the production blockchain. • Build pipeline(s) include consortium defined tests • Deployment pipeline(s) driven by approved consortium policies

Local development environment

A project within Azure DevOps is created and source control will either be provided by Azure DevOps (Azure Repos) or an external Git service such as Github.

A developer is creating or editing local copies of one or more smart contracts. The developer will deploy, test, and debug against a standalone node.

The developer will have automated tests and may also do local UI testing. In some cases, they may also do other types of tests in a standalone fashion, e.g. interactions with code hosted in compute, such as serverless offerings like Functions, Logic Apps or Flow, that is distinct to that developer.

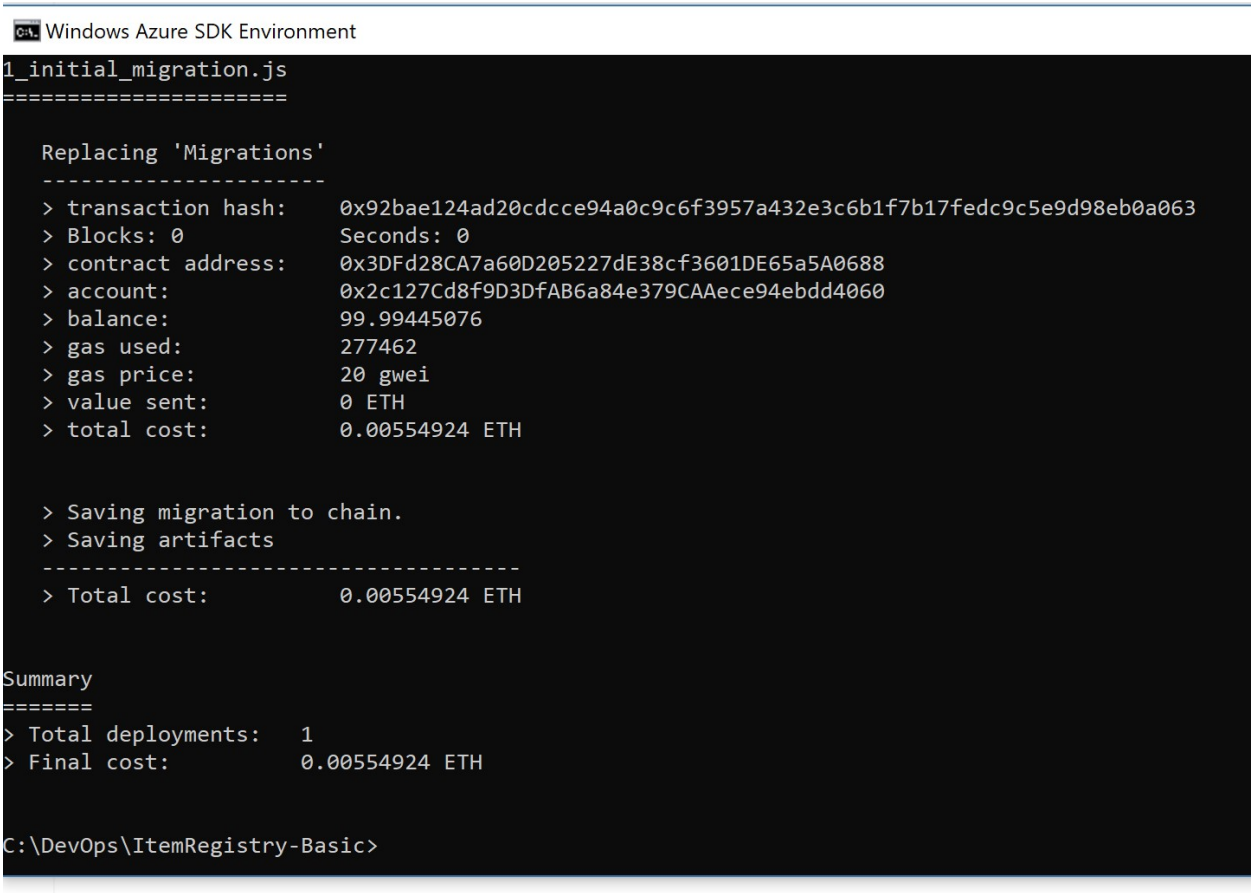
On the Ethereum blockchain, these are typically written as javascript or solidity and executed using Truffle. The two commands used include "truffle compile", which compiles contracts and "truffle test"

Once tests are passed the developer will submit a pull request to their git repo.

Based on policy, the pull request will be reviewed by the designated number of reviewers established and accepted as appropriate into source control.

Source control will either be provided by Azure DevOps (Azure Repos) or an external Git service such as Github.

Deployment of contracts for an Ethereum blockchain can be done with the "truffle migrate" command.



Automated Testing

Truffle enables the development of automated tests using javascript or solidity.

The implementation guide focuses on two contracts, BasicItemRegistry and Item.

The BasicItemRegistry test demonstrates how to achieve the automated testing goals defined earlier in this paper. This is done using Truffle's embedded mocha test runner. The top-level test suite is defined within a **`contract`** callback. This lets Truffle prepopulate the test with a reference to your contract's constructor so that you can easily deploy new instances of the contract for each test. Within the contract callback, multiple levels of **`describe`** callback are used to drill down into each area of the contract under test. Within each **`describe`**, one or more **`beforeAll`** or **`before`** callbacks set up and assert the preconditions for each test. Finally, one or more **`it`** callbacks define the test cases, which contain the action under test and assertions to validate the expected postconditions of that action.

It is possible Truffle tests to have one test case depend on the post condition of another. **This should be avoided.** Test execution time is a common reason for developers designing tests this way, but usually this is a sign of undesirable system complexity and that issue should be addressed before proceeding.

You'll notice in the implementation guide that each contract state under test is set up for each test case independently of the rest. This is done for two reasons. First, it more explicitly expresses the contract author's intent to developers or auditors. This on its own saves large amounts of time when tests fail. Second, it makes it possible for developers to run each test case without needing to run any of the

others, which makes it as easy as possible to troubleshoot test failures with a debugger.

Test Against Production Blockchain Data Locally with Forking

Good DevOps processes attempt to eliminate or minimize the difference between test environments and production. Blockchains facilitate this goal in a way that other software cannot via the ability to fork a chain. Forking a chain is a process where a new blockchain network is created from the state of an existing chain, but blocks added to the new chain do not get appended back to the original. With tools like ganache-cli, instantaneous forking can be achieved by deferring back to the original network for reads of old state information.

Setting up a forked network is easy using the ganache-cli. This is done by pointing the ganache-cli at the RPC endpoint of the node you wish to fork by running it with the --fork flag. If you are using a private chain for a consortium or the public Ethereum network, the approach is the same.

An example command line would be the following -

ganache-cli --fork https://mynode.mynetwork.local:8545

Once the above command is executed, the ganache-cli will expose its own local network on port 8545 which will refer back to the original chain for reads of state that existed prior to the fork. There is no delay for this to happen, Ganache will be available immediately.

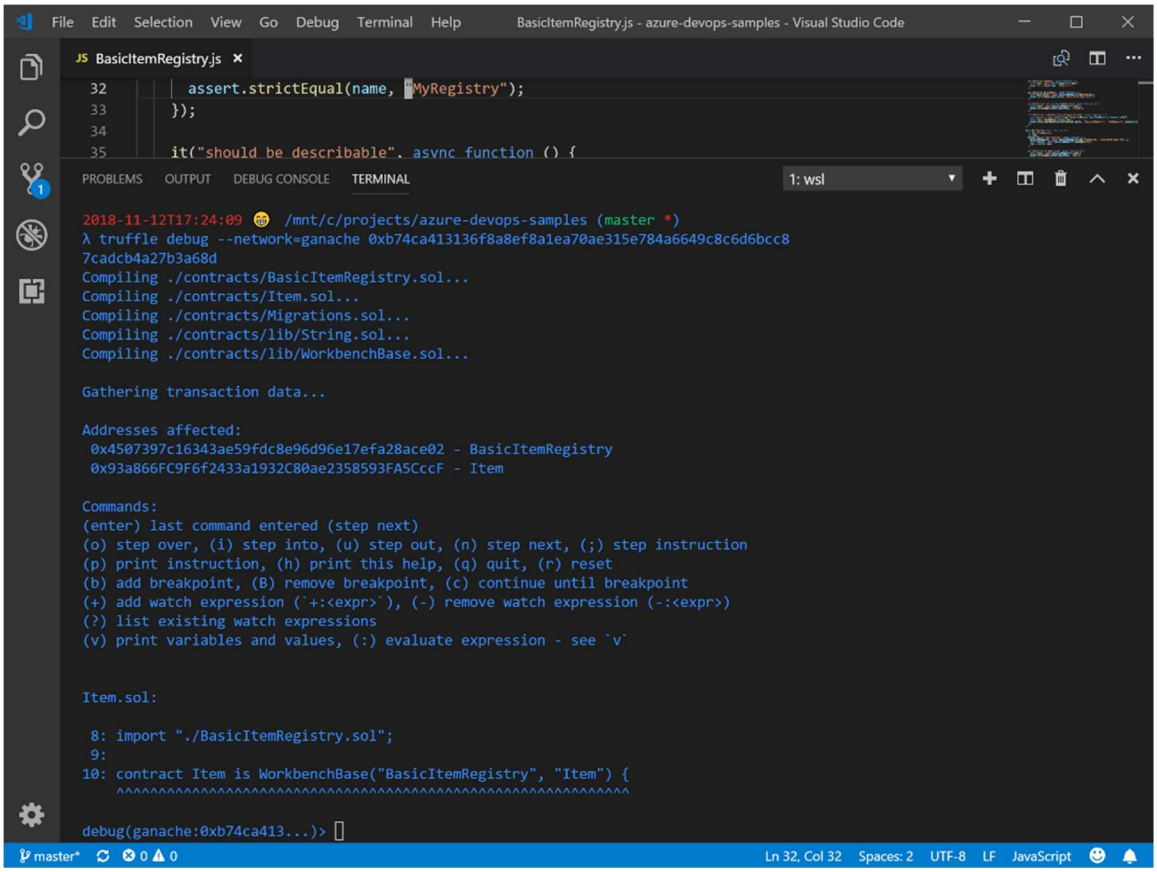
Debugging with Truffle

Truffle offers a command-line Solidity debugger which allows you to inspect the execution of a transaction after it has executed. Executing the debugger is done in three steps. First, execute the transaction you wish to debug. Second, copy the hash for the transaction you just ran. Third, run truffle debug $HASH – where $HASH is the transaction hash you recorded in the previous step.

An example command line would be the following -

truffle debug 0x5f522794c73eab4be53c26f9855edd115d615f95d3e7074a3cec52c16b1daa84

If successful, you should see a screen like the one below. On-screen instructions identify how to step, add breakpoints and watches, and to inspect variable state.



Note – If debugging against a network other than the default “development” network be sure to include the --network flag, as shown in the screenshot above.

Submit a pull request to check this in with the consortium repo

Once debugging has been successfully completed, code is checked in as it normally would be.

Consortium member test environment

A consortium member test environment is intended to be used for development and testing by a team working on internal applications that use the smart contract(s) *at a specific consortium member*

This environment has the following characteristics –

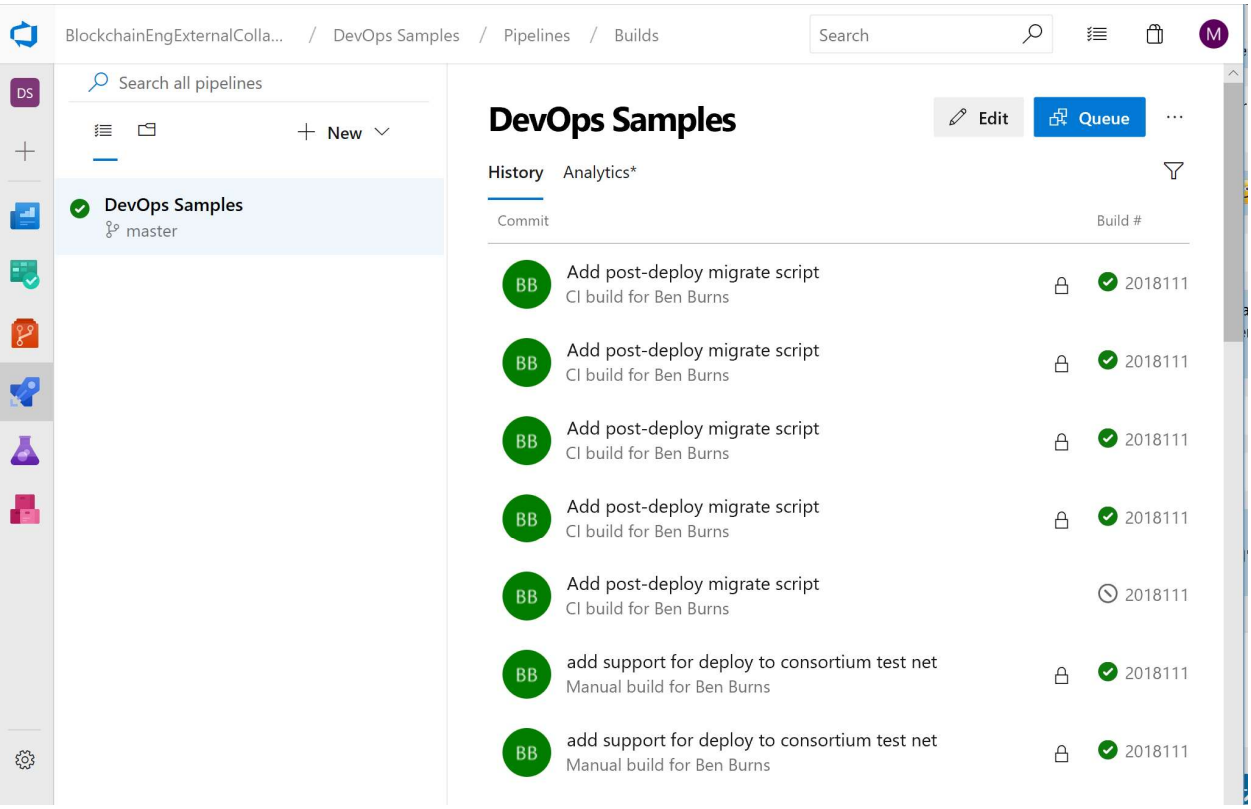
- Infrastructure is ephemeral or permanent
- Test and deployment are done against a “forked chain” that represents production.
- Build pipeline(s) include includes consortium and consortium member defined tests for smart contracts and dependent applications
- Deployment pipeline(s) driven by approved consortium and consortium member policies

Implementing for Ethereum VM based blockchains

In this section you will use a mix of Azure Pipelines and Truffle to build, test, and deploy your smart contracts.

Build Pipeline

A Build Pipeline in Azure Pipelines is used to compile and test the code. The build pipeline will incorporate Truffle to compile and test the smart contracts.



The sample project for the implementation guide was included in the Repo and includes an azure-pipelines.yaml file.

If you examine the file, you will see that it performs the following activities -

- Installs Node.js
- Installs NPM
- Runs truffle compile
- Runs truffle test
- Installs Node.js dependencies
- Installs Mocha
- Publishes test results
- Copies files (post build)
- Publishes build artifacts

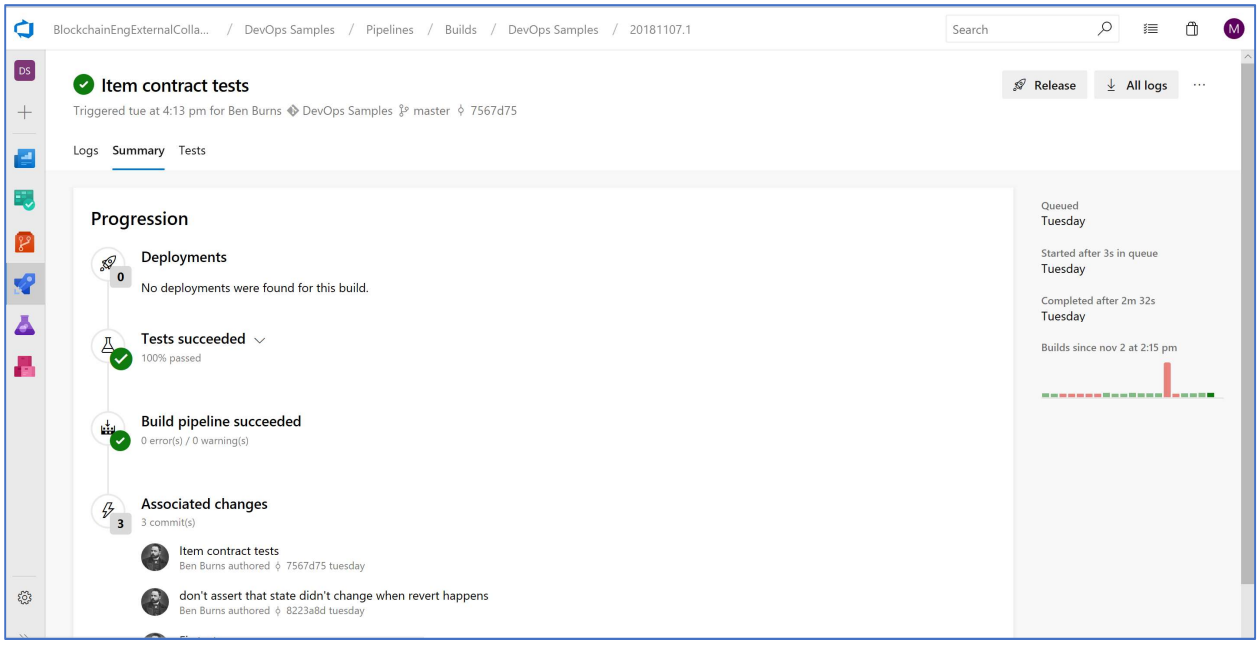
Automated Tests

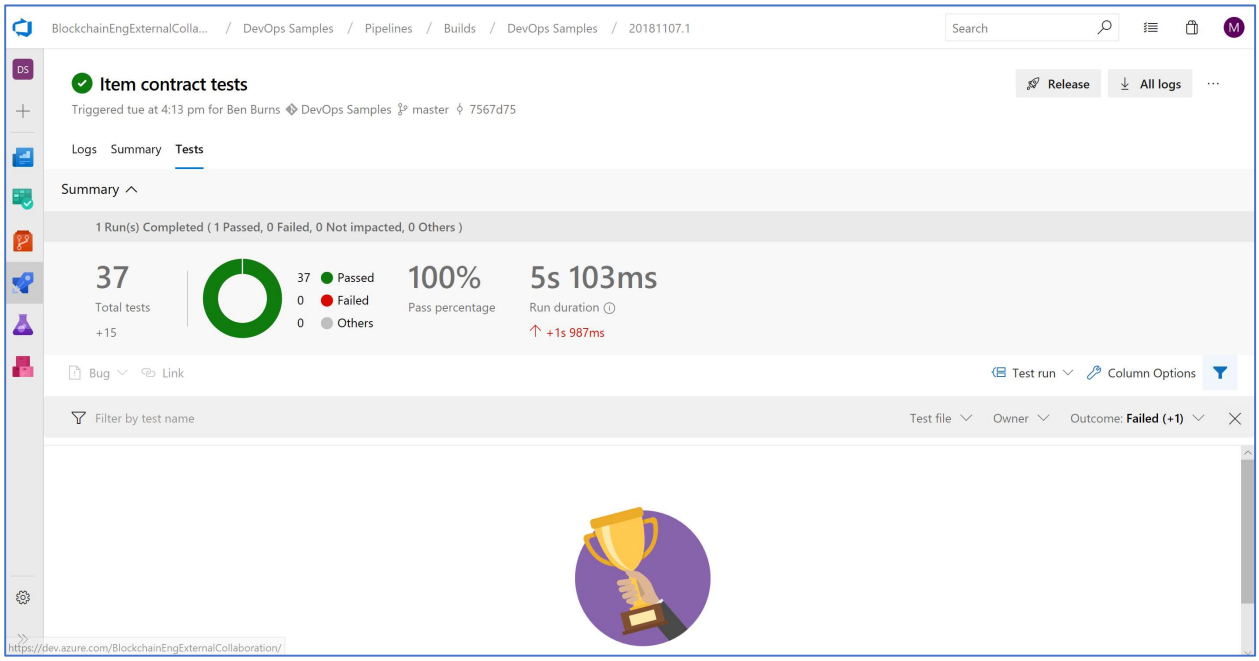
The same unit tests that were performed locally are performed automatically when code is checked into the repo.

After a build pipeline is executed, select the build in the Builds section of Azure Pipelines to show the outcome of the tests. Azure Pipelines will identify –

- What changes were incorporated in the build
- Whether the build was successful (identifying any errors that may have occurred)
- Whether the tests were successful (identifying any failed tests that may have occurred)
- What deployments were triggered as a result of these tests.

The Summary and Test pages for a build of the sample project can be found below.





At the consortium member level, there will be custom code that interacts with the smart contract. This can be code running in compute (VM, Logic Apps, Flow, Functions, etc.) or a web app communicating directly using the native blockchain API endpoints or client libraries, e.g. Nethereum, that wrap them.

This is consortium specific code that may manifest itself in the communication of data and events to and from the smart contract from backend systems, web or mobile clients for end users, etc.

The consortium member should determine what tests should be written to validate the smart contracts have not changed in a way that will break compatibility with their existing integration points. These should then be incorporated in build and deployment pipelines as appropriate.

Upgrades and Testing in an Immutable Blockchain

In some use cases, smart contracts represent processes that may span many days, weeks, months, even years. A key consideration for testing is understanding the requirements of releasing a new version of a contract to an immutable blockchain and incorporating those requirements into your testing.

If a contract has two versions, e.g. 1 and 2, a determination must be made as to how existing versions of the contract should be handled. In some cases, existing "in flight" smart contracts will continue to function and in other cases they will need to migrate to a newer version of the contract.

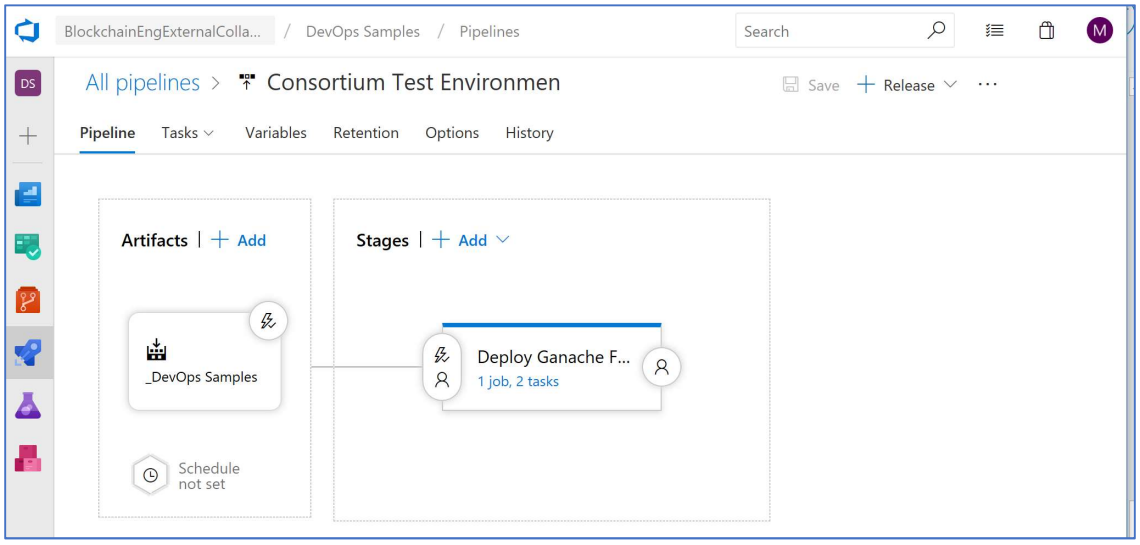
For example, if a bank were to introduce a new loan process that had new requirements, it may only want to apply that to new loans or loans whose term is greater than 180 days. In other cases, legislation, industry regulation, or corporate policies may change how a process must be handled in every situation.

Based on the requirements of the scenario, tests that include upgrading existing contracts, compatibility with applications across multiple versions of contracts, etc. should be considered and added as appropriate.

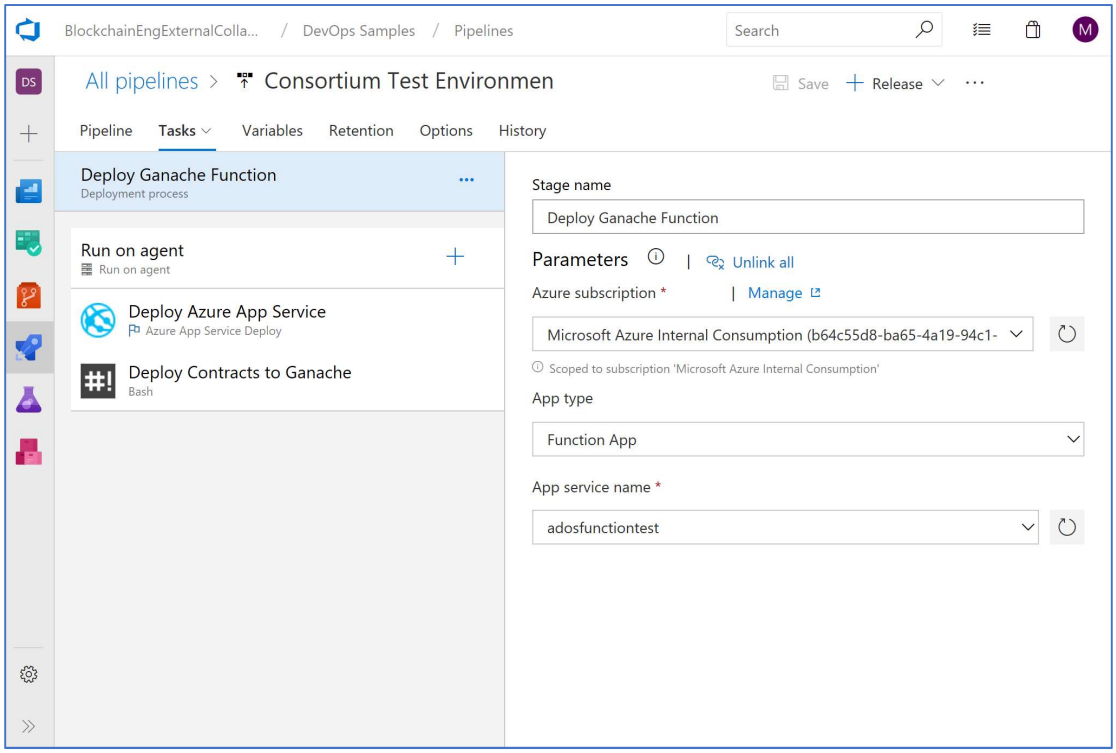
Once testing is successfully completed, a release pipeline can copy files and publish build artifacts for use in a release pipeline.

Release Pipeline

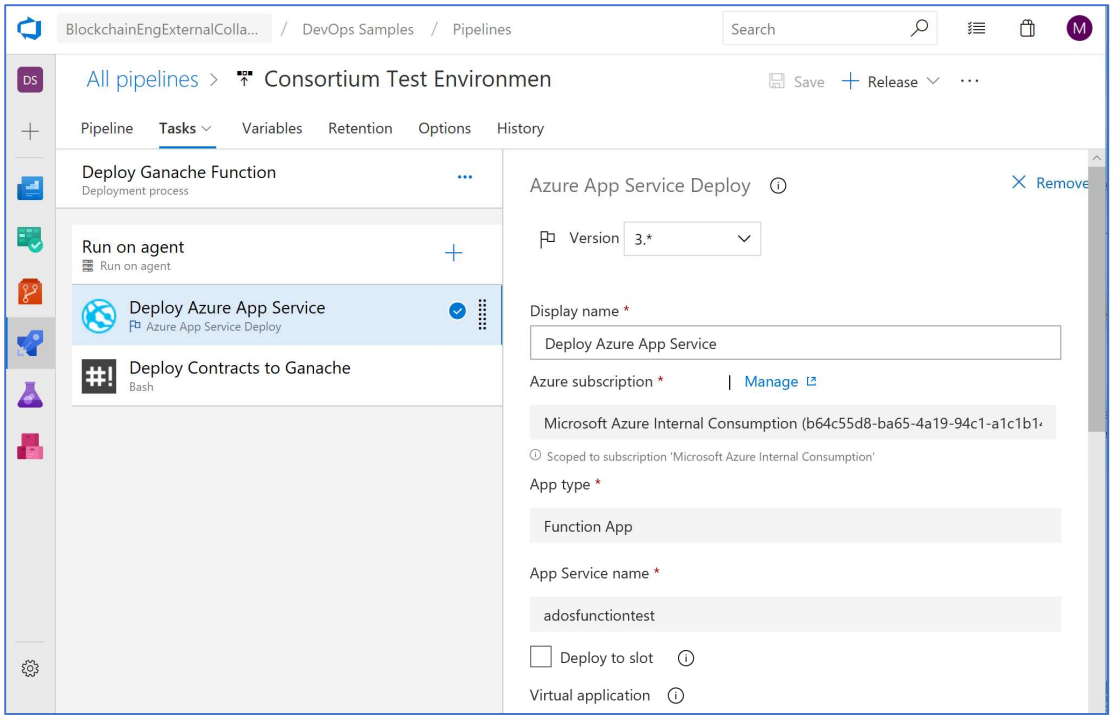
A Release Pipeline will deploy the smart contracts into an environment where they can be tested. For testing, deployment can be to Ganache or a full deployment of an Ethereum ledger done as infrastructure as code.



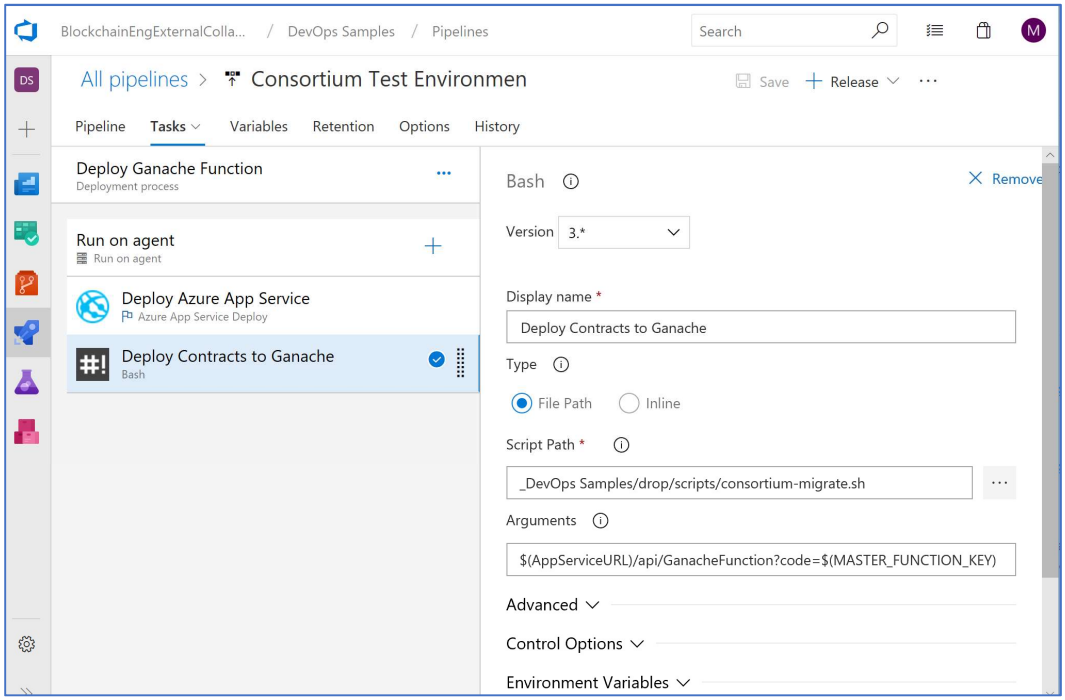
In this paper we will focus on using Ganache in a function as it is a lower cost option that still represents a production chain. For public chains, this is the most appropriate option. For private chains, whether you use Ganache or a deployment of a ledger, e.g. Quorum, is a decision to be made by the consortium member.



The deployment pipeline will have two tasks – one will deploy the function with Ganache.



The other will use a bash script to deploy the smart contracts that were compiled and tested in the build pipeline.



To deploy a full ledger network or node via infrastructure as code, an Azure Resource Manager (ARM) template can be used. Additional details on this are provided in the implementation guide.

When defining your release pipeline(s), you should also incorporate the dependent applications so that the resulting test environment represents production for the consortium member.

Upgrade requirements and scenarios reflected in testing, if any, should also be incorporated into the release pipeline

Consortium test environment

A consortium test environment is intended to be used for development and testing by the cross-consortium team working on consortium applications that use the smart contract(s) in the project.

This environment has the following characteristics –

- Infrastructure is ephemeral or permanent
- Test and deployment are done against a "forked chain" that represents production.
- Build pipeline(s) include consortium approved tests for contracts and dependent applications
- Deployment pipeline(s) driven by consortium targeted release environment and approved consortium policies

Build Pipeline

The core build pipeline for the consortium will resemble the core of what has been used at the consortium member.

At the consortium level, there will also be custom code that interacts with the smart contract. This can be code running in compute (VM, Logic Apps, Flow, Functions, etc.) or a web app communicating directly using the native blockchain API endpoints or client libraries, e.g. Nethereum, that wrap them.

This is consortium specific code that impacts all members of the consortium and may manifest itself in the communication of data and events to and from the smart contract from backend systems, web or mobile clients for end users, etc.

Build pipelines should incorporate this additional code as appropriate.

Automated Tests

The consortium should determine what tests should be written to validate the smart contracts have not changed in a way that will break compatibility with their existing integration points.

If build pipelines for these dependent applications do not already exist, then these additional tests should then be incorporated in build pipelines.

As with the consortium member test environment, when defining your release pipeline(s), you should also incorporate the dependent applications so that the resulting test environment represents production for the consortium member.

Upgrade requirements and scenarios, if any, should also be reflected in testing

Release Pipeline

A Release Pipeline for the consortium will deploy the smart contracts into an environment where they can be tested. For testing, deployment can be to Ganache or a full deployment of an Ethereum ledger done as infrastructure as code.

As with the deployment pipeline for the consortium member, in this paper we will focus on using Ganache in a function as it is a lower cost option that still represents a production chain. For public chains, this is the most appropriate option. For private chains, whether you use Ganache or a deployment of a blockchain, e.g. Ethereum, is a decision to be made by the consortium member.

Note – some chains, e.g. Quorum, are based on Ethereum but have extended the functionality of it. Ganache does not yet support this extended functionality and infrastructure as code should be used to represent the chain in these cases.

In addition to the smart contracts and the ledger, there are often consortium specific applications that must be tested to ensure compatibility and identify any breaking changes. In the previous section, this was done for applications specific to a consortium member, e.g. Contoso's back end applications. In this instance, what will be deployed are the applications provided for the entire consortium that have a dependency on this smart contract, if any.

Upgrade requirements and scenarios reflected in testing, if any, should also be incorporated into the release pipeline

Infrastructure as Code

As in previous sections, we will utilize Ganache to provide a forked version of a production chain.

For some consortiums, there is a desire to deploy infrastructure as code that represents the topology and specific ledger stack that they will deploy in production. For the very first deployment, there is no notion yet of production, and it may also be desirable to deploy a full node or network.

In these cases, the deployments should be consistent, predictable and with an infrastructure as code approach.

The recommended approach is to use an automation template. On Azure, this is done using an Azure Resource Manager (ARM) template.

Consortium Production Environment

A consortium production environment is intended to be used by the members of a consortium for consortium applications and downstream dependent applications of consortium member applications that use the smart contract(s).

This environment has the following characteristics –

- Infrastructure is permanent
- Deployment is done against a production blockchain.
- Build pipeline(s) include consortium approved tests for contracts and dependent applications
- Release pipeline(s) driven by consortium policies

Build Pipeline

The build pipeline for production should mirror the build pipeline for the Consortium Test Environment.

Automated Tests

Automated tests for production should mirror the tests done for the Consortium Test Environment.

Release Pipeline(s)

The deployment pipeline(s) should mirror those that were used in the Consortium Test Environment with some exceptions.

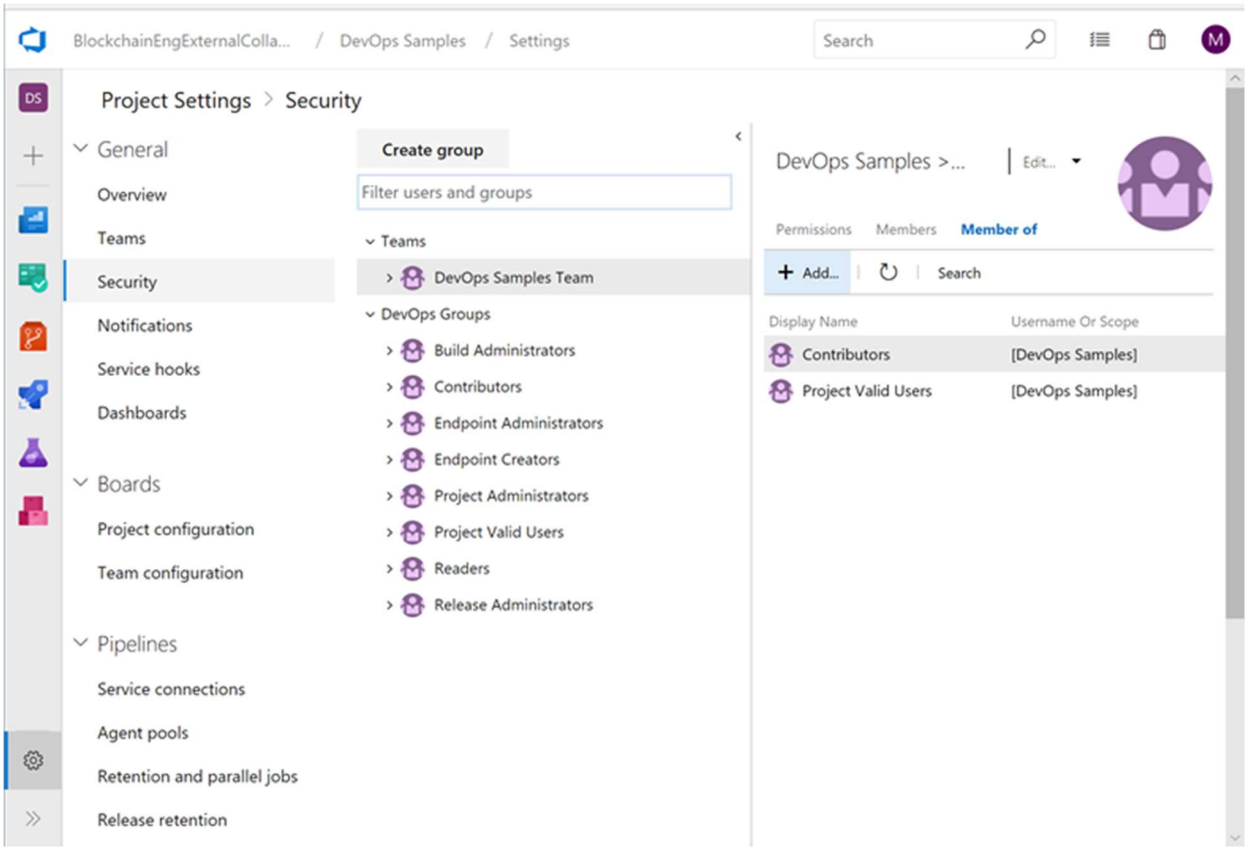
In a test deployment pipeline, a new instance of a dependent application may be deployed for user acceptance testing. In production, that application may already exist and if there are no changes to the application code, it may be inappropriate or unnecessary to redeploy the application.

Unlike other environments where deployment may happen at will, in a production environment where there is a dependency on shared data within a blockchain the release(s) should be coordinated with members of the consortium to ensure there is no business impact to consortium members.

Business impact could result from timing of releases. A release for a global consortium could occur during a time of day where there is high use of dependent applications, a holiday where there is no staff on hand to support, could require coordination of consortium members to deploy updates to their own applications to be compatible with the new smart contract versions, or other impacts from rolling out changes that would upgrade contracts and require some coordination with consortium members.

Branch Security

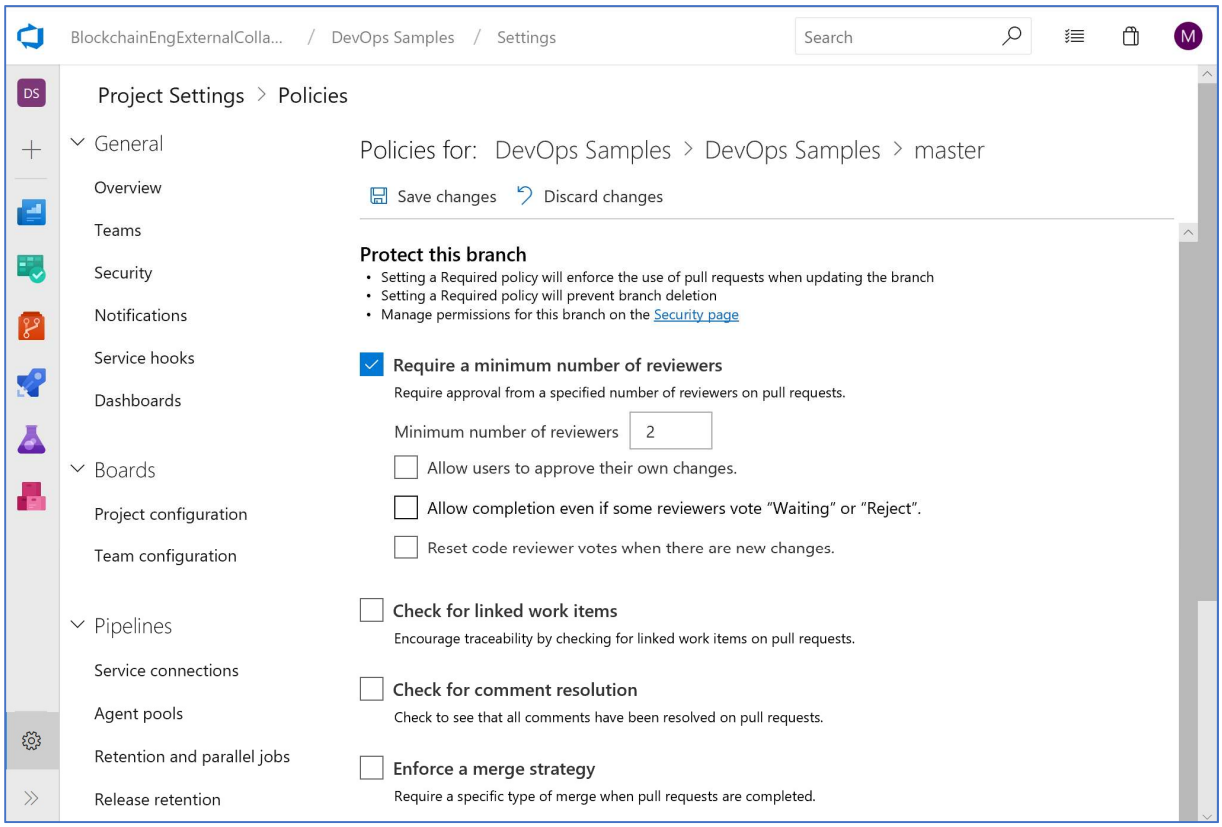
Branch security should be configured for role based access control. It also enables placing individuals into groups which can be used in branch policies (described below). Azure DevOps supports users from multiple organizations and aligns well to the multi-party needs of a consortium.



Branch Policies

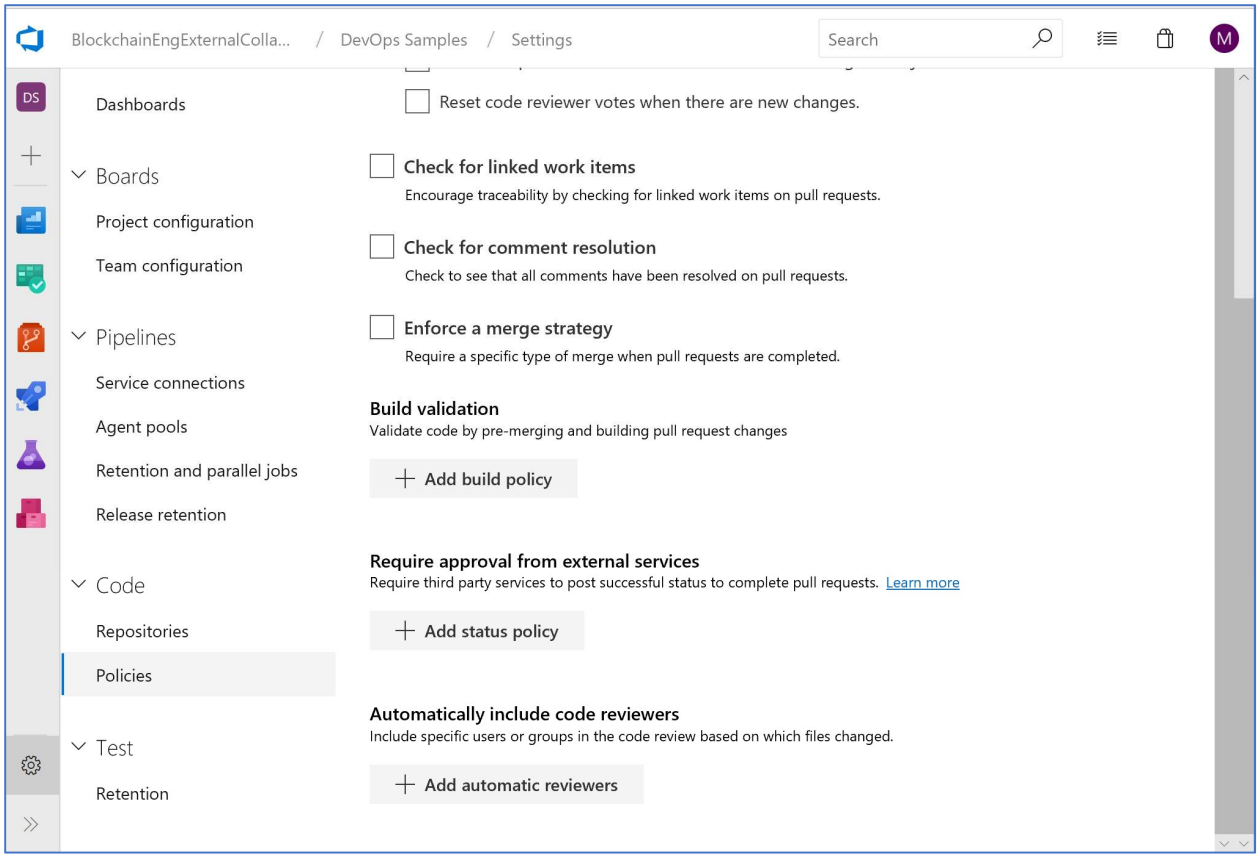
Azure DevOps provides the ability to assign branch policies.

For consortiums, the most important policies are tied to the assignment of reviewers. DevOps provides the ability to specify the number of reviewers required and a specific pool of reviewers required based on the files changed.



Policies can require that pull requests are tied to work items. This helps ensure code creation or changes are tied to work items that are identified, assigned, and tracked by the consortium.

Policies can also require approval from external services which can be helpful in a consortium environment.



Conclusion

While there are certain considerations related to blockchain technology and a consortium model, DevOps for blockchain is readily achievable with existing tools such as Azure DevOps, VS Code and Truffle.

To implement the approach outlined in this paper there is an implementation guide with detailed instructions.