Ensuring Deterministic Execution in Modern Language Runtimes

A never-ending journey for correctness

What We'll Cover

- How Temporal defines determinism and the 3 common violations
- Approaches to ensure determinism in modern language runtimes
  - Includes challenges/approaches for Go, JVM, Node.js, Python, and .NET
  - Approaches graded on "difficulty" - difficulty to implement, maintain, and use
  - Approach graded on "effectiveness" - how well it works
  - Grades are letter-based - A is best, F is worst
- Extra topics (time permitting)
- AI/LLM-based code analysis, Hosted WASM, Replaying, Fuzzing

Temporal’s Definition of Deterministic Execution

- Workflows must be **deterministic**
- General definition - same code path and same effects for same input
- Temporal definition - same commands for the same input
    - Workflows, based on events, produce a series of commands
    - On replay, Temporal expects same series of commands
    - Different code paths technically ok if same commands produced
    - Not every part of command checked (e.g. activity input can change)
- Deterministic execution required so workflows can replay/resume elsewhere

The 3 Common Determinism Violations

Determinism Violation #1 - Illegal Calls

- Many language calls are inherently non-deterministic. Often includes:
  - **System time** - any kind of tickers, timers, current date fetching, deadline-based timeouts
  - **Random** - including any form of non-deterministic cryptography
  - **IO** - disk, network, stdio, etc
  - **Threading** - thread pools, OS threading, non-deterministic coroutine/event/channel scheduling, etc
  - **Hash map iteration** - often includes sets and ones with deterministic-yet-unpredictable iteration
- Common prevention approaches:
  - **Emulation** - replacing calls, though can confuse developers with existing expectations
  - **Static analysis** - best approach, though often requires opt-in and not fool-proof
  - **Prevention** - patching/prevention at runtime, would prefer earlier in process

Determinism Violation #2 - Shared State Mutation

- Most modern languages inherently encourage mutable shared state
- Mutable state ok within workflow, bad across workflows
- Includes externally provided immutable state, e.g. configuration, that may differ across workers
- Common prevention approaches:
  - **Sandbox/interpreter** - Global/shared state still exists, but it's isolated. Can have performance implications. Advanced cases may still need to punch holes.
  - **Static analysis** - Not great for this use case. Often unreliable in dynamic languages. Hard to tell whether shared state is still deterministic (e.g. memoization).
  - **Runtime interception** - Most runtimes don't offer clear hooks at this level

Determinism Violation #3 - Code Changes

- Code changes affecting command output/order are non-deterministic when replaying older workflows
- Unreasonable for most modern tools to diff two sets of workflows
- Workflow versioning approaches are out of scope for this talk
- For this presentation, only concerned that human can obviously see non-deterministic changes
- Means if developer changes code that doesn't look to change command order/output, it shouldn't

  Problematic areas include:
  - Deterministic collection iteration - includes deterministically stable yet unpredictable iteration
  - Unpredictable internal coroutine reordering
  - Internal SDK logic changing from one SDK version to the next

Approaches for Non-Determinism Prevention in Go

A runtime with very little flexibility
but clear semantics

Challenges Posed by Go

- Cannot use any concurrent construct, they are all non-deterministic
- No easy path to altering scheduling to make concurrent constructs deterministic
- Map iteration is intentionally non-deterministic
- All normal standard library limitations apply (e.g. time, random, etc) with no simple way to prevent at runtime
- Language is very opinionated about disallowing and/or making it difficult to do runtime or compiled code alterations
- Temporal has written deterministic alternatives for unusable native constructs
- Temporal has written a static analyzer to catch most issues, but not all

Go Approach #1 - Static Analysis

- Temporal built **workflowcheck**, a static analyzer that transitively checks for non-deterministic constructs
- Needs to be improved to analyze shared state mutation
    - Hard because referencing global vars in Go is normal and many times safe
- Go is easy to statically analyze and it's not as common to avoid analyzable paths (e.g. reflection) as it is in other languages
- Go programming language changes runtime libraries from version to version which may trip the detector if not maintained constantly
- Grade:
    - Difficulty: **B**
    - Effectiveness: **B**

Go Approach #2 - Interpreter

- Good for illegal calls, shared state mutation, and code changes
- Can make goroutines work deterministically and prevent illegal calls
    - Do not want to emulate illegal calls to avoid developer expectations concerns
- Existing interpreters like **Yaegi** and the **SSA version in /x/** have shortcomings and cannot be customized well
- Language is easier to write interpreter for compared to most others, but still very time consuming and a high maintenance cost
- Slight performance cost, but negligible for workflows
    - Some packages need to be non-interpreted for performance (e.g. expensive-to-initialize unicode package)
- Only catches problems at runtime
- Grade:
    - Difficulty: **D**
    - Effectiveness: **B**

Go Approach #3 - Transpiling

- Have written **open source POC** that wraps go build for code alterations
  - Designed to only be adjacent to unaltered code
  - Can be leveraged for altering Go constructs to make them deterministic
- Heavy effort altering code to make deterministic
  - Have to maintain debuggability and get variable hygiene/scoping right
- Heavy effort isolating global state
  - Have to alter all package field access/mutation to essentially use goroutine-locals
- Requires user opt-in to a manual compilation step
- Grade:
  - Difficulty: **C-**
  - Effectiveness: **A**

Other Go Approaches

- Fork Go
  - Obviously a maintenance nightmare
  - Grade: Difficulty: **F**, Effectiveness: **B**
- Contribute to Go
  - First Google result for “golang deterministic” is **issue opened by Temporal CEO** - discussed and closed by Go maintainers
  - Very reasonable that Go team does not want to maintain this rare/unique use case
  - Grade: **Nonstarter**
- Runtime tracing
  - Monkey patching is brittle and infects non-workflow use of the runtime
  - eBPF is (mostly) platform dependent and also would infect non-workflow use of the runtime
  - No reasonable approach helps with shared state mutation
- Grade: Difficulty: **F**, Effectiveness: **C**

Approaches for Non-Determinism Prevention on the JVM

A runtime with lots of flexibility, but is moving faster than some tools can keep up

Challenges Posed by the JVM

- Same common sources of non-determinism exist in standard library
- Multiple languages are common, approaches may need to apply generally
- Most runtime-based approaches operate on bytecode but are hard to differentiate from non-workflow code in a performant way

JVM Approach #1 - Static Analysis

- Checkstyle provides path forward here
  - Often only used for Java, leaving Scala/Kotlin/Clojure/etc out
  - JDK already annotated with purity annotations like @Deterministic, but their rules are much too strict
- Tools like Semgrep or others could also be leveraged instead
- Will have to maintain known JDK non-deterministic calls, but not impossible even if not 100% accurate
- Like Go static analysis, may be too many false positives on shared state mutation
- Grade:
  - Difficulty: **B**
  - Effectiveness: **B**

JVM Approach #2 - Interpreter/Sandbox

- Many metacircular interpreters for the JVM
  - Older ones don't support modern major classfile bytecode versions
  - Will need to research Graal's Espresso interpreter to see if configurable enough
- Existing deterministic runtimes (e.g. Corda) are not generic for our use
- Cannot use JVMTI-based stepping approach, performance drops too dramatically
- Will have to prevent threading, not feasible to just emulate
- Performance drop due to there being so many standard library instructions to interpret
  - Go which has a lot fewer
  - These paths won't use JIT'd hot paths
  - Could defer to outside the interpreter, but complicated to share objects across the boundary
- Can write JVM interpreter (there aren't that many instructions) but heavy effort
- Grade:
  - Difficulty: **C**
  - Effectiveness: **B**

JVM Approach #3 - Runtime Tracing

- Agents can instrument easily to prevent calls
  - Bytecode injection difficult to
  - Requires bytecode injection at these points, which is fine for direct workflow code, but is unreasonable for shared code
- Code instrumented at class loading time, so cannot tell whether code called from workflow or not
  - Multi-class-loader approach for same classes has many issues (despite OSGi and other efforts)
  - Too expensive to do thread local checks in shared code at each injection point
- Can do compile-time instrumentation + shading, but shading has too many caveats to do automatically
- Field mutations tracked per workflow (heavy) for shared state mutations
- Due to the above, best approach is workflows in separate JVM, negates some benefits
- Grade:
  - Difficulty: **C**
  - Effectiveness: **B**

Approaches for Non-Determinism Prevention in Node.js

The whole language/runtime was built to be sandboxed, so no problem

Node.js Approach #1 – V8 Sandbox

- What we do today, works very well for obvious reasons
  - Prevents shared state mutation
  - Allows deterministic promise ordering
- Language never had threading, no system-level concurrency concerns
- JS standard library small, non-deterministic aspects easily disabled/emulated
- Temporal bundles code to ensure isolation
- Memory concerns when completely reloading/isolating each run, but many are addressable
- Grade:
  - Difficulty: **A-** (implementation difficulty - but we've already done the work)
  - Effectiveness: **A**

Node.js Approach #2 - Static Analysis

- Sandbox already exists, why is this needed?
  - Ensure query handlers and update validators have no side effects (but we don't want an entire V8 context/isolate)
  - Can dry-run bundle step at this build-time step to catch other issues
- Limited effectiveness with dynamic JS, but can do good enough
- Grade:
  - Difficulty: **B**
  - Effectiveness: **B**

Approaches to Non-Determinism Prevention in Python

A runtime that’s so flexible,
can it be constrained yet still powerful?

Challenges for Python

- Flexible asyncio lets us have a deterministic coroutine scheduling
  - Some pieces like IO or deadline-based timeouts have to be disabled
- No good sandbox exists, but the whole thing is a reusable interpreter anyways
  - Custom importers - complex
  - Restricting parts of the standard library - complex
  - Reloading all modules for every run - performance/caching concerns
- Non-deterministic standard library functions are (mostly) obvious
- C extensions are ubiquitous and cannot be isolated reasonably
  - Modern C extensions are expected not to use shared state, older ones do

Python Approach #1 - Interpreter/Sandbox/Tracing

- No good existing sandbox exists for our use case
- We wrote our own
  - Custom importer - loading transitive modules, but allows “pass through” from outside for performance
  - Memory efficient - passes through most modules
  - Some C extensions (e.g. older protobuf) don’t support reloading due shared global state
  - Restricted standard library calls proxied - subtle Python bugs here
  - Proxying builtins like open() requires actually patching with a thread-local check
  - Maintain known list of illegal standard library calls
  - Does help prevent global state sharing inside the same file
- Only good at runtime, so fails late in process
- Other implementations like RustPython and PyPy no more flexible for our use case
- Grade:
  - Difficulty: **C** (implementation difficulty - but we’ve already done the work)
  - Effectiveness: **B**

Python Approach #2 - Static Analysis

- Pyre, MyPy, etc based static analyzer is very doable
- We already maintain a curated list of illegal calls
- Coupled with sandbox, can probably avoid shared state mutation checks
- Since type hints needed/encouraged anyways, static analyzer fairly accurate
  - User can easily fall back to untyped, dynamic Python that is hard to check
  - Most benefit is on illegal calls, easy to find even untyped
- Grade:
  - Difficulty: **B**
  - Effectiveness: **B**

Approaches to Non-Determinism Prevention in .NET

A semi-flexible runtime with hidden surprises and opinionated approaches

Challenges for .NET

- .NET has coroutines as tasks, but they mix with threaded use
- Can customize TaskScheduler, but inconsistently ignored sometimes
  - .NET static analyzer rules even encourage avoiding custom TaskSchedulers
- One inflexible global static thread pool
  - Hard to know when transitive code is using it
- Reasonable in-process sandboxing approaches have been abandoned
  - AppDomain and Code Access Security not implemented in modern .NET
  - Maintainers afraid of insecure sandboxes though we just need any sandbox
- Timers are not interceptable
  - Anything with timeout uses threading and/or system timers - all non-deterministic
  - Just merged TimeProvider API can help newer .NET versions
- Great support for static analysis in ecosystem
- Temporal .NET is young - only limited research has been done

.NET Approach #1 - Runtime Tracing

- Implemented today
  - Too many surprise implicit threading and system timer use in standard library
  - Uses EventListener invoked on every task to check proper scheduler
  - Task events listened process wide, though performantly ignored if outside of workflow
- Only applies at runtime
  - So your workflow suspends when a bad call is made
- Does not help shared state mutation or most illegal calls
- Grade:
  - Difficulty: **B**
  - Effectiveness: **B-**

.NET Approach #2 – Static Analysis

- Great support for static analysis, many projects provide their own analyzers
  - Users much more likely to use than other more opt-in languages
- Easy to customize in editorconfig, supported well by IDEs
- Works well against illegal calls
  - Obviously reflection works around, but not that common
  - Includes all task calls that we runtime trace for
- Works ok against shared state mutation
  - Some transitive dependencies' mutations may be allowed that are deterministic
  - For example, hard to know whether shared Lazy's Value getter is deterministic
- Grade:
  - Difficulty: **B**
  - Effectiveness: **A-**

.NET Approach #3 - Interpreter/Sandbox

- With AppDomain and Code Access Security no longer available
  - No reasonable in-process sandboxing approaches
- Can recompile CIL at runtime
  - Large effort to do transitively, brittle, large effort to isolate state mutations
- Researching whether Mono or other CIL interpreters are flexible enough
- Can create interpreter
  - Large effort
  - Can't use expression trees, too many unsupported language features
- Many users may not be ok with the performance penalty
- Grade: **Researching**

General Approaches for Non-Determinism Prevention

A couple of language-agnostic approaches that don’t work as well as they may seem

General Approach #1 - System-level Sandbox/Isolation

- Good for illegal calls and shared state mutation, bad for code changes and developer ergonomics
- Recent tools like **Hermit** help provide deterministic OS primitives
- Low-level abstractions violate our humans-can-predict-code-path-changes rule
    - Deterministic OS threading too low level, cannot know how slight change can alter coroutine order
    - Shared seeded OS-level random hard to predict use of by language code
- Still must disable/emulate system time to avoid false expectations (e.g. monotonicity)
- Often requires containerization - adds runtime overhead and complexity
- Grade:
    - Difficulty: **C**
    - Effectiveness: **C**

General Approach #2 - Self-Interpreted WASM

- Good for illegal calls and shared state mutation, bad for code changes and developer ergonomics
- Low-level abstractions violate our humans-can-predict-code-path-changes rule
- No (current) Temporal language has good, small WASM code gen
  - **Go** - bloated WASM, compile-time opt-in, unclear coroutine reordering, limited call interception flexibility, alternative Go implementations incomplete
  - **JVM** - bloated WASM, no blessed WASM compiler (4 competing, Graal waiting on WASM GC), JVM bytecode better than WASM bytecode for this use, performance drop even with JIT WASM runtime
  - **Node.js** - already sandboxed, no need
  - **Python** - usually just interpreter as WASM, no benefit over native interpreter
  - **.NET** - Blazor is web specific, usually not CIL as WASM but runtime that can use unmodified CIL which is promising
- Grade:
  - Difficulty: **C**
  - Effectiveness: **B-**
- **But!** Hosted WASM has real value, touched on later

Verdicts

What Will Be Implemented?

- **Go**
  - Static analysis - already done, will improve
  - All other options - unlikely in near term if ever
- **JVM**
  - Static analysis - very likely
  - All other options - unlikely in the near term if ever
- **Node.js**
  - Sandbox - already done, performance will continue to improve
  - Static analysis - possible if valuable enough for query handlers and update validators
- **Python**
  - Sandbox - already done, ergonomics are still rough though
  - Static analysis - very likely
  - All other options - unlikely in the near term if ever
- **.NET**
  - Runtime tracing - already done, but of limited value
  - Static analysis - very likely
  - All other options - unlikely in the near term if ever

Additional Ideas

Replayer - Finding Non-Determinism After the Run

- Replaying existing runs helps find non-deterministic code
- Replays can help vet new code
- Good CI setups will replay all workflows (or at least a lot) from history that should be compatible with code
    - Can also download/build a corpus of workflow histories to check during unit testing
- Replays also have other value for debugging

A Hosted WASM Future

- Presentation so far centered on workflow workers in runtime workflows/activities
- Users that are willing to run in a different runtime, WASM provides a great option
  - Stepping stone to supporting FaaS/serverless workflows
  - Stepping stone to supporting server-side hosted workers
  - All non-determinism concerns in this presentation resolved
- Ideal implementation:
  - Rust WASM runner on top of our Rust core
  - IDL of acceptable calls for import to do workflow things
  - Per-language shims (or leverage abstractions like **Extism**) to make writing workflows easier
  - Possible introduction of a workflow primitive for executing HTTP activities to cross the workflow/activity boundary more easily
- This is a very active area of research and development at Temporal, stay tuned

AI/LLM Powered Static Analysis - GPT4 Prompt

Temporal.io is a Workflow Orchestration solution that requires users to adhere to a set of platform specific rules in order to benefit from the value Temporal provides. One of the core rules of the platform is that code written in a Temporal Workflow must be deterministic, which means that you cannot use standard I/O and system library calls that are themselves non-deterministic. A clear example of this would be a call that sleeps on some timer, as a system timer is implemented based on some underlying processor heuristics that are not deterministic. Fortunately Temporal provides special purpose primitives in each language which provide functionality which is traditionally non-deterministic in a deterministic way. For example there is a durable sleep call provided by workflows that is safe to use.

I would like you to validate whether the following workflow breaks the rules of Temporal. If it does, please respond with a programmatically consumable artifact that details all instances of rule breakage (at minimum with correspond line number and rule breakage type).

AI/LLM Powered Static Analysis – GPT4 – The Bad Code

```
time.Sleep(1000 * time.Second)
```

```
log.Println("some text")
```

```
package greetings

import (
  "time"
  "log"
  "go.temporal.io/sdk/workflow"
)


func GreetingSample(ctx workflow.Context) (string, error) {
  logger := workflow.GetLogger(ctx)

  ao := workflow.ActivityOptions{
    StartToCloseTimeout: 10 * time.Second,
  }
  ctx = workflow.WithActivityOptions(ctx, ao)

  var a *Activities // use a nil struct pointer to call activities that are part of a structure

  var greetResult string
  err := workflow.ExecuteActivity(ctx, a.GetGreeting).Get(ctx, &greetResult)
  if err != nil {
    logger.Error("Get greeting failed.", "Error", err)
    return "", err
  }

  time.Sleep(1000 * time.Second)

  // Get Name.
  var nameResult string
  err = workflow.ExecuteActivity(ctx, a.GetName).Get(ctx, &nameResult)
  if err != nil {
    logger.Error("Get name failed.", "Error", err)
    return "", err
  }

 log.Println("some text")

  // Say Greeting.
  var sayResult string
  err = workflow.ExecuteActivity(ctx, a.SayGreeting, greetResult, nameResult).Get(ctx,
&sayResult)
  if err != nil {
    logger.Error("Marshalling failed with error.", "Error", err)
    return "", err
  }

  logger.Info("GreetingSample completed.", "Result", sayResult)
  return sayResult, nil
}
```

AI/LLM Powered Static Analysis - GPT4 - Result

The provided Temporal workflow breaks the rules of the platform in two instances, specifically by using a non-deterministic call and a standard I/O call. Below is the programmatically consumable artifact that details the rule breakages:

json Copy code

```
[
  {
    "line_number": 24,
    "rule_breakage_type": "non_deterministic_call",
    "description": "Usage of time.Sleep is not allowed in Temporal workflows
  },
  {
    "line_number": 34,
    "rule_breakage_type": "standard_io_call",
    "description": "Usage of log.Println is not allowed in Temporal workflow
  }
]
```

1. The non-deterministic call is the usage of `time.Sleep` on line 24, which is not allowed in Temporal workflows as it relies on system timers and can cause non-deterministic behavior. To fix this issue, replace the `time.Sleep` call with `workflow.Sleep`:

go Copy code

```
workflow.Sleep(ctx, 1000*time.Second)
```

2. The standard I/O call is the usage of `log.Println` on line 34, which is not allowed in Temporal workflows as it can cause non-deterministic behavior. To fix this issue, replace the `log.Println` call with `logger.Info`:

go Copy code

```
logger.Info("some text")
```

These changes will ensure that the workflow adheres to the rules of the Temporal platform and behaves deterministically.

json Copy code

```
[
  {
    "line_number": 24,
    "rule_breakage_type": "non_deterministic_call",
    "description": "Usage of time.Sleep is not allowed in Temporal workflows
  },
  {
    "line_number": 34,
    "rule_breakage_type": "standard_io_call",
    "description": "Usage of log.Println is not allowed in Temporal workflow
  }
]
```

AI/LLM Powered Static Analysis - Notes

- An active area of research
- Can this be used today?
  - Sure! Take similar prompt and use your GPT4 access if you have it
  - GPT3.5 is considerably worse at detection
  - Does not help transitive code checking, OpenAI token/context limitations prevent large scale checking
- Challenges for Temporal to make a tool using this
  - GPT4 expensive for this use case and of limited value for large codebases (even if dissected)
  - Ideally large, open source models will emerge that we can fine tune for Temporal's rules and use case
  - Too hard to predict if/when the non-OpenAI world will catch up on quality
- Why would I ever hand-write a static analyzer again?
  - Spending weeks hand-writing rules that traverse over typed ASTs starting to seem like a waste of time
  - Ecosystems like .NET analyzers still need to work within their framework
  - If open source, self-hosted AI ecosystem continues blistering pace, a general purpose workflow-mistake-catching tool written (and/or hosted) by Temporal may be just around the corner
- We would love to collaborate here! Get in touch!

Fuzzing Workflows

- Active area of research at Temporal
- Approach:
  - Fuzzer emulates all possible events given workflow shape and commands generated so far
  - Each branch is separate on a different fuzzer path
  - Customization can include expected/unexpected results
- Can leverage burgeoning language runtime fuzzer support
  - For example, easy to have helper accept workflow, some configuration, and *testing.F in Go
- Has to be built SDK-side in process
  - Server-side event generation fuzzing is just not fast enough
- While generally applicable to find bugs, can also find non-determinism
  - Often non-determinism occurs in code paths not often traveled

The End - Ideas Welcome!

- Any technologies/approaches for preventing non-determinism are welcome
- Most ideas can be implemented on top of SDKs (e.g. a workflow that invokes a sandbox) to just try out ideas
- At the least, this is a fun problem (we're hiring! **https://temporal.io/careers)**
- Get in touch:
  - Slack - **https://t.mp/slack**
  - Forums - **https://community.temporal.io/**

Build invincible apps

It’s never a bad time to be invincible