



BazelCon 2026

Persistent Test Runners

BazelCon 2026

John Cater, Senior Software Engineer @ EngFlow

Antonio Di Stefano, Founder @ Frekt.dev

#BazelCon



Intro: Persistent Test Runners

What is a Persistent Test Runner?

Persistent Workers — but for tests.

It's that simple.

Eliminates repeated overhead by reusing long-running test execution processes across tests and builds.

Examples of overhead:

- Process startup: fork/exec, JVM initialization
- Heavyweight fixtures: external databases
- Container image pulls



Benefits of Persistent Test Runners

Eliminates Process Startup Overhead

Bypasses recurring JVM, emulator, and simulator initialization times by keeping execution processes active across builds.

Accelerates Build & Test Times

Drastically reduces feedback loops and developer wait times by continuously processing incoming test execution requests.

Leverages JIT Compilation Caches

Reuses long-running execution environments to benefit from warm JIT compilation state instead of cold-starting target code.



History of `--experimental_persistent_test_runner`

2019 – 2022

ADDED & ROLLED BACK

Added flag and test runner
Rolled back the same day

COMMITTS

9700ee677870
1a4e55dab13b

RE-ADDED

The flag was
re-added

COMMIT

02f8da948bf3

Jan 11, 2020

Mar 21, 2022

Dec 6, 2019

ADDED

Re-added just the
persistent test runner
code

COMMIT

6fff77a5b0e4

Jan 14, 2020

REMOVED

The flag and test
runner were removed

COMMIT

1f33504af3eb



BazelCon 2026

Problems with the Original Implementation

Buggy & Unstable: Worked initially but suffered from critical bugs in real-world scenarios.

Classpath Handling

Didn't handle classpath changes well during incremental builds.

Stale classes and state leaks corrupted test runs.

Tight Coupling

Relied heavily on `java_test` being a native rule.

Embedded Java test runner inside Bazel, blocking extensibility.

JIT Compilation Overhead

In the original implementation, significant time was spent JIT compiling target code, which could not be avoided per run for correctness.

Shared Dependencies

Common dependencies between the test runner and code under test (e.g. Guava) made separation difficult.

JVM State Purging

Extremely hard to purge previous JVM state across test runs without complex classloader isolation.



Persistent Test Runners in Bazel



BazelCon 2026

Current Test API & Example Test Rule

Test API

- Create an executable and return it in `DefaultInfo`
- This executable has all needed configuration, takes no flags
- Environment based on Test Encyclopedia
- Exit code determines pass/fail
- Stdout becomes the test log
- Test can emit its own test.xml **or** Bazel will create one based on the test log

Extensive use of environment variables for setup:

- Sharding
- Test filtering
- Available tmpdir

Example Test Rule Definition

Creating a custom test rule in Starlark:

```
def _custom_test_impl(ctx):
    exe = ctx.actions.declare_file(ctx.label.name)
    ctx.actions.write(
        output = exe,
        content = "#!/bin/bash\nexit 0\n",
        is_executable = True,
    )
    return [
        DefaultInfo(executable = exe),
    ]

custom_test = rule(
    implementation = _custom_test_impl,
    test = True,
)
```



Worker API & Example Action

Worker API Overview

- Action is created via `ctx.actions.run` with specific `execution_requirements`
- Execution requirements tell Bazel persistence is supported
 - The worker protocol (protobuf or JSON)
- Whether multiplexing is supported
- Gives a worker key to improve reuse

Execution Persistence

Bazel decides whether the action is actually persistent

- Potentially persistent actions must support both single-shot and persistent execution
- In either case, action-specific data is provided via a paramfile

Example Worker Action Definition

Creating a persistent worker action in Starlark:

```
worker_tool = ctx.executable._tool
args = ctx.actions.args()
# Add target-specific args to args
args.use_param_file("@%s", use_always = True)
ctx.actions.run(
    executable = worker_tool,
    arguments = [args],
    inputs = [input_file],
    outputs = [output_file],
    execution_requirements = {
        "supports-workers": "1",
        "supports-multiplex-workers": "1",
        "requires-worker-protocol": "json",
        "worker-key-mnemonic": "MyWorker",
    },
)
```



A New Proposal

An API for persistent tests

- Designed to seamlessly integrate with modern Bazel execution models.
- Works directly with the `Starlark test` and `worker APIs`.

Full proposal coming soon

We will present a full proposal for community discussion and design based on the proof-of-concept described here.



PersistentTestInfo Provider

Provider Definition

Defined provider structure for persistent test execution:

```
test_info = PersistentTestInfo(  
    ctx = ctx,  
    multiplex = False,  
    requires_worker_protocol = protocol,  
    worker_key_mnemonic = "CaseCheckWorker",  
    worker_executable = case_checker,  
    worker_tools = [tools],  
    worker_args = ["--protocol=" + protocol],  
    test_inputs = [first_file, second_file],  
    test_args = [test_args],  
)  
  
return [  
    DefaultInfo(),  
    test_info  
]
```

Execution Requirements

The provider has similar fields to the `execution_requirements` from `ctx.actions.run`:

- `multiplex`
- `requires_worker_protocol`
- `worker_key_mnemonic`

Action Data Fields

And some fields that resemble data from `ctx.actions.run`:

- `worker_executable`
- `test_inputs` and `worker_tools`
- `worker_args` and `test_args`



Persistent Test Runners in rules_java



BazelCon 2026

Integrating with rules_java (Again)

Pre-warm the next JVM

A coordinator runs multiple JVMs: 1 for the running test and 1+ for the future tests that get warmed up in the meantime.

Pros:

- Instead of blindly reusing a JVM, the coordinator checks a test's dependencies to decide whether a JVM can be reused

Cons:

- Significant coordination overhead and complexity
- Requires instrumenting the test JVMs

Using fork...

Only the calling thread survives. JVM locks and runtime state may depend on threads that disappeared, making continued Java execution unsafe.

... Using Zygote ...

Zygote is Android's pre-warmed virtual machine process. This might work, but it only exists on the Android Runtime (ART), not on the JVM.

... and in general ...

When should we checkpoint? Finding the right moment requires test-writer help, framework integration or guessing.

... but the coordinator works!

Despite these issues, the coordinator idea works: it analyzes each test's dependencies, decides whether a JVM can be reused, and injects the test.



Reusing the JVM

First, we check if a JVM can be reused for a given test:

Same startup configuration?

The coordinator checks the Java executable, JVM options, driver code and plugin configuration.

Existing jars unchanged?

Every jar used within the lifetime of a JVM must retain its checksum. If the next test requests the same jar identity, its checksum must match too.

New dependencies?

New jars extend the child's classloader and become part of future compatibility checks.

Then, reuse a compatible child, or start another JVM. Update the test's execution context, call JUnit and return the result, keeping the JVM alive for the next test.

But we also need to make sure that we don't poison a test with additional dependencies:

The coordinator and child communicate through a custom protocol built using Java's standard library. This avoids introducing transport libraries that could conflict with the test's dependencies.



Runner Lifecycle and Extensions

How to implement

Implement an extension class and annotate it:

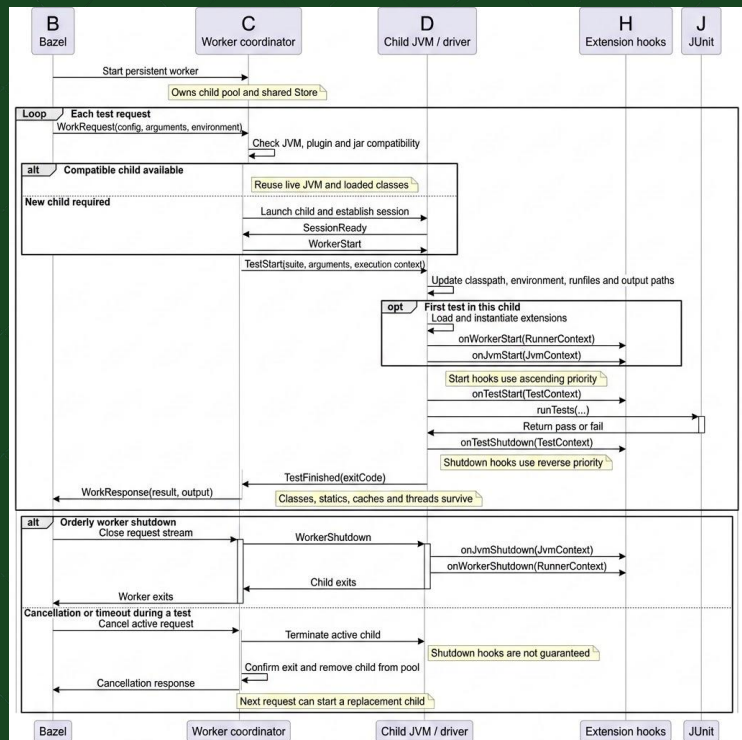
```
@PersistentTestRunnerExtension
public final class MyHooks implements Extension {
    public void onJvmStart(JvmContext ctx) {}
    public void onTestStart(TestContext ctx) {}
    public void onTestShutdown(TestContext ctx) {}
    public void onJvmShutdown(JvmContext ctx) {}
}
```

onJvmStart and **onJvmShutdown** are called once during the lifetime of a JVM.

onTestStart and **onTestShutdown** are called once for each test execution, so 1+ times on each JVM.

Extensions run on the test's JVM, not the coordinator.

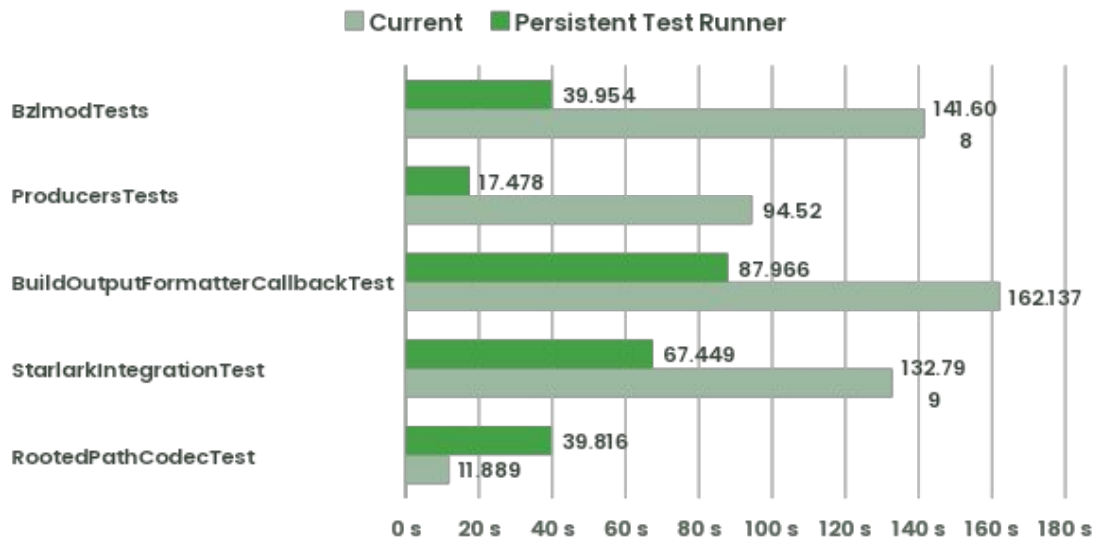
Extensions are automatically discovered, just add them as dependencies of a test target or a test library.



Java Test Results: 46% Faster

Selected targets: median runtime

Bazel's own Java tests: local strategy vs. persistent worker



Overall

Total of median runtimes across 46 Bazel Java test targets, 4 runs each

46%

less total test time

Current **36.5 min**

Persistent **19.6 min**

Saved **16.8 min**

35 of 46 targets got faster. The 11 slower ones are all short tests (under 14 s).

How Can I Try It Out?

Cloning the repos

Bazel:

```
git clone --branch persistent-test-workers \
  https://github.com/TheGrizzlyDev/bazel.git
```

rules_java:

```
git clone --branch persistent-test-workers \
  https://github.com/TheGrizzlyDev/rules_java.git
```

Build Bazel:

```
bazel build //src:bazel-dev; \
  cp bazel-bin/src/bazel-dev /path/to/custom/bazel
```

Running tests with Persistent Test Runners

```
/path/to/custom/bazel test \
  --override_module=rules_java=/path/to/rules_java \
  --enable_persistent_test_runners=JvmTestRunner \
  --strategy=TestRunner=worker,local \
  //path/to:tests
```



Experimental

This feature is under active development. Don't use it in production yet.



BazelCon 2026

Next Steps

1 Proposal Finalization

Finalize the proposal and share it with the community for feedback and design discussion.

2 Prototype Cleanup

Clean up the existing Bazel prototype to prepare it for wider review and testing.

3 Code Review

Send the cleaned up prototype for review.

4 Rule Updates

Make changes to test rules to adopt and use the new persistent testing functionality.

- Other JVM Languages
- iOS and Android tests
- Large integration tests (test containers)

