Here’s a tight, concrete plan that keeps reuse precise and safe, based on your current tree.

Why `context` needs special handling

`AnalysisContext` exposes things that *do* affect results (at least):

- `declaredVariables` — feeds constant evaluation. You call it today from `element.dart` via `library.context.declaredVariables` (3 call sites).
- `sourceFactory.dartSdk` — used by `ErrorVerifier` to decide whether a re-exported `dart:` library is “internal”.

If we just slap `@trackedDirectlyOpaque` on a `context` getter, *every* call through it will produce an `OpaqueApiUse`, and you’ll lose reuse across a lot of diagnostics/const work. So the right approach is:

1. add **precise, tracked accessors** for the parts we actually use (`declaredVariables`, and a tiny helper for the SDK “internal” check), migrate internal uses to them; **then**
2. expose `context` via an **opaque** getter to catch any remaining/external use without silently missing it.

Below are surgical diffs and the matching requirements wiring.

Step A — Precise accessors + migrate internal uses

1) `LibraryElementImpl`: add tracked accessors; don’t touch `context` yet

File: `analyzer/lib/src/dart/element/element.dart`

Add imports once (top of file, where other public analyzer imports live):

```
diff

+import 'package:analyzer/dart/analysis/declared_variables.dart';
```

Inside `class LibraryElementImpl` add:

```
diff

   // TODO(scheglov): replace with `LibraryName` or something.
   int nameLength;

+  /// Precisely tracked access to the current -D variables snapshot.
+  @trackedDirectly
+  DeclaredVariables get declaredVariables {
+    globalResultRequirements?.record_library_declaredVariables(element: this);
+    return context.declaredVariables;
+  }
+
+  /// A narrow helper for ErrorVerifier: whether a `dart:` URI denotes an
+  /// internal SDK library. Keeps us off of `context` for this hot path.
+  @trackedDirectly
+  bool isInternalSdkLibrary(Uri uri) {
+    globalResultRequirements
+        ?.record_library_isInternalSdkLibrary(element: this, uri: uri);
+    final sdk = context.sourceFactory.dartSdk;
+    final lib = sdk?.getSdkLibrary(uri.toString());
+    return lib?.isInternal ?? false;
+  }
```

Migrate the 3 const-evaluation call sites to the precise accessor:

```diff
-         declaredVariables: library.context.declaredVariables,
+         declaredVariables: library.declaredVariables,
```

(Do this for all three occurrences in **`element.dart`**.)

2) `ErrorVerifier`: use the narrow helper instead of walking through `context`

File: **`analyzer/lib/src/generated/error_verifier.dart`**

Replace the internal check:

```diff
-    // should be private
-    var sdk = _currentLibrary.context.sourceFactory.dartSdk!;
-    var uri = exportedLibrary.uri.toString();
-
-    var sdkLibrary = sdk.getSdkLibrary(uri);
-    if (sdkLibrary == null) {
-      return;
-    }
-    if (!sdkLibrary.isInternal) {
-      return;
-    }
+    if (!_currentLibrary.isInternalSdkLibrary(exportedLibrary.uri)) {
+      return;
+    }
```

This eliminates the *internal* consumers of **`context`** that would otherwise cause opaque requirements.

Step B — Now expose `context` opaquely

With internal call sites moved off **`context`**, we can safely gate **external** use.

File: **`analyzer/lib/src/dart/element/element.dart`**

Change the field to private and add a tracked getter:

```diff
-class LibraryElementImpl extends ElementImpl
+class LibraryElementImpl extends ElementImpl
     with DeferredResolutionReadingMixin
     implements LibraryElement {
-  final AnalysisContext context;
+  final AnalysisContext _context;
```

Update the constructor tuple:

```diff
-  LibraryElementImpl(
-    this.context,
+  LibraryElementImpl(
+    this._context,
     this.session,
     this.name,
     this.nameOffset,
     this.nameLength,
     this.featureSet,
   );
```

Add the getter:

```diff
+  @trackedDirectlyOpaque
+  AnalysisContext get context {
+    globalResultRequirements?.recordOpaqueApiUse(this, 'context');
+    return _context;
+  }
```

> This preserves behavior for existing users (`library.context` still works), **records** any remaining access as opaque, and keeps our own code using the precise accessors.

Step C — Requirements wiring (precise, non-opaque)

You need two small pieces of state in `RequirementsManifest`:

- a snapshot of `-D` **declared variables** (names and values),
- the set of `dart:` URIs you asked about via `isInternalSdkLibrary`.

File: `analyzer/lib/src/fine/requirements.dart`

Add fields to `RequirementsManifest`:

```diff
 class RequirementsManifest {
   ...
   final List<OpaqueApiUse> opaqueApiUses = [];
+
+  /// Global environment snapshot used during this result.
+  /// (null if not read)
+  Map<String, String>? declaredVariables;
+
+  /// Any `dart:` URIs for which we asserted "is internal SDK lib".
+  Set<String> requiredInternalSdkUris = {};
```

Add recorders:

```diff
   RequirementsManifest();
   ...
+  void record_library_declaredVariables({required LibraryElementImpl element}) {
+    if (_recordingLockLevel != 0) return;
+    // Snapshot only once.
+    if (declaredVariables == null) {
+      final dv = element.context.declaredVariables;
+      final map = <String, String>{};
+      for (final name in dv.variableNames.toList()..sort()) {
+        map[name] = dv.get(name) ?? '';
+      }
+      declaredVariables = map;
+    }
+  }
+
+  void record_library_isInternalSdkLibrary({
+    required LibraryElementImpl element,
+    required Uri uri,
+  }) {
+    if (_recordingLockLevel != 0) return;
+    requiredInternalSdkUris.add(uri.toString());
+  }
```

Serialize/deserialize (after existing lists; keep order consistent):

```
diff

   factory RequirementsManifest.read(SummaryDataReader reader) {
     var result = RequirementsManifest();
     ...
     result.opaqueApiUses.addAll(
       reader.readTypedList(() => OpaqueApiUse.read(reader)),
     );
+    result.declaredVariables = reader.readOptionalObject(() {
+      return reader.readMap(
+        readKey: () => reader.readStringUtf8(),
+        readValue: () => reader.readStringUtf8(),
+      );
+    });
+    result.requiredInternalSdkUris = reader
+        .readOptionalObject(() => reader.readTypedList(reader.readStringUtf8))
+        ?.toSet() ??
+        {};
     return result;
   }

   void write(SummaryDataWriter sink) {
     sink.writeMap(
       libraries,
       writeKey: sink.writeUri,
       writeValue: (v) => v.write(sink),
     );
     sink.writeTypedList(exportRequirements, (e) => e.write(sink));
     sink.writeTypedList(opaqueApiUses, (e) => e.write(sink));
+    sink.writeOptionalObject(declaredVariables, (map) {
+      sink.writeMap(
+        map,
+        writeKey: sink.writeStringUtf8,
+        writeValue: sink.writeStringUtf8,
+      );
+    });
+    sink.writeOptionalObject(
+      requiredInternalSdkUris.isEmpty ? null : requiredInternalSdkUris.toList(),
+      (list) => sink.writeTypedList(list, sink.writeStringUtf8),
+    );
   }
```

Verify early in **`isSatisfied(...)`**:

```
diff

   RequirementFailure? isSatisfied({
     required LinkedElementFactory elementFactory,
     required Map<Uri, LibraryManifest> libraryManifests,
   }) {
     if (opaqueApiUses.isNotEmpty) {
       return OpaqueApiUseFailure(uses: opaqueApiUses);
     }
+
+    // Check declared variables snapshot (global).
+    if (declaredVariables != null && libraryManifests.isNotEmpty) {
+      final firstUri = libraryManifests.keys.first;
+      final lib = elementFactory.libraryOfUri(firstUri);
+      final dv = lib?.context.declaredVariables;
+      final current = <String, String>{};
+      if (dv != null) {
+        for (final name in dv.variableNames.toList()..sort()) {
+          current[name] = dv.get(name) ?? '';
+        }
+      }
+      if (!const MapEquality<String, String>().equals(declaredVariables, current)) {
+        return DeclaredVariablesMismatch(
+          expected: declaredVariables!,
+          actual: current,
+        );
+      }
+    }
+
+    // Check that all URIs we asserted as internal are still internal.
+    if (requiredInternalSdkUris.isNotEmpty && libraryManifests.isNotEmpty) {
+      final firstUri = libraryManifests.keys.first;
+      final lib = elementFactory.libraryOfUri(firstUri);
+      final sdk = lib?.context.sourceFactory.dartSdk;
+      for (final s in requiredInternalSdkUris) {
+        final ok = sdk?.getSdkLibrary(s)?.isInternal ?? false;
+        if (!ok) {
+          return SdkInternalLibraryExpectationFailed(uriString: s);
+        }
+      }
+    }
```

Add simple failures:

```diff
 class OpaqueApiUseFailure extends RequirementFailure { ... }
+
+class DeclaredVariablesMismatch extends RequirementFailure {
+  final Map<String, String> expected, actual;
+  DeclaredVariablesMismatch({required this.expected, required this.actual});
+  @override
+  String toString() => 'DeclaredVariablesMismatch';
+}
+
+class SdkInternalLibraryExpectationFailed extends RequirementFailure {
+  final String uriString;
+  SdkInternalLibraryExpectationFailed({required this.uriString});
+  @override
+  String toString() => 'SdkInternalLibraryExpectationFailed($uriString)';
+}
```

(Place these alongside the other small `RequirementFailure` subclasses you already have.)

Step D — Add focused tests (no golden content needed)

Add to `FineAnalysisDriverTest`-style suite:

- **Const + -D variables**
  - `test_manifest_declaredVariables_const_initializer_reuses_when_same`

    - **`test_manifest_declaredVariables_const_initializer_relinks_when_changed`**
    - Scenario: a top-level **`const int k = int.fromEnvironment('X', defaultValue: 1);`**
        - First run: no variables; second run: set **`X=2`**; expect reuse fails with **`DeclaredVariablesMismatch`**.
- **SDK internal check**
    - **`test_manifest_isInternalSdkLibrary_stable_for_dart_core`** (asserts reuse)
    - **`test_manifest_isInternalSdkLibrary_fails_when_sdk_changes`** (simulate by switching to a mocked SDK bundle where **`dart:_internal`** flips — you already have SDK test scaffolding; otherwise just assert the requirement is recorded.)
- **Opaque guard**
    - A small plugin/test that *reads* **`library.context`** directly (e.g., via a custom verify hook) should yield an **`OpaqueApiUseFailure`** and no reuse; this proves the guard works.

You said you generate expectations automatically, so names + scenarios are enough.

Summary

- Add *precise* tracked accessors: **`declaredVariables`** and **`isInternalSdkLibrary(Uri)`**.
- Migrate internal uses off **`context`**.
- Then expose **`context`** via **`@trackedDirectlyOpaque`** getter (backed by a private field) to catch any remaining/external use.
- Teach **`RequirementsManifest`** to (a) snapshot **`-D`** variables and (b) remember the set of **`dart:`** URIs asserted as “internal”, with verification.
- Add three small test clusters to lock behavior.

This gives you correctness (env changes never silently reuse) **and** precision (constant evaluation & the SDK internal export check no longer poison reuse via an opaque requirement), while still guarding untracked external use of **`context`**.

❞ Ask ChatGPT