SATE AI: A Fault Injection Framework for On-Device AI Models in Flutter

Muhammad Assad Ullah

1 August 2026

Department of Computer Science, University of Engineering and Technology, Taxila, Pakistan
ORCID: 0009-0007-2290-304X

Summary

SATE AI is an open-source fault injection and stress-testing framework for on-device artificial intelligence (AI) models running inside Flutter applications. As machine learning inference increasingly moves from the cloud onto phones and embedded devices, developers must contend with a class of failure modes that conventional software tests do not exercise: memory pressure during inference, malformed or adversarial input, numerical drift from post-training quantization, thermal throttling of the host CPU, latency degradation under load, corrupted or swapped model files, and silent confidence collapse. SATE AI packages these failure modes as a library of composable *fault injectors* that can be run against any on-device model through a common adapter interface, producing a structured pass/fail report that developers can inspect locally, export, or wire into continuous integration (CI).

The framework ships with seven fault injectors (`MemoryPressureInjector`, `MalformedInputInjector`, `QuantizationDriftInjector`, `ThermalThrottleInjector`, `LatencyInjector`, `ModelSwapInjector`, and `ConfidenceThresholdInjector`) and three model adapters (`MockAdapter` for pure-Dart testing without a real model, `OnnxAdapter` for ONNX Runtime models, and `TFLiteAdapter` for TensorFlow Lite models). Results are exposed through a `StressRunner` and `Report` API that supports JSON, Markdown, and CSV export, a browser-based dashboard with dark mode and mobile-responsive layout for visual inspection, a command-line interface for local and scripted runs, and a GitHub composite action so that stress tests can gate pull requests the same way unit tests do.

SATE AI is implemented in Dart, distributed via pub.dev, and covered by 164 automated unit tests spanning all core modules. As of this writing it has reached pub.dev's maximum quality score of 160/160 and nine published releases (v0.1.0 through v0.7.0), reflecting an iterative design process driven by real usage on on-device large language model (LLM) and embedding-model deployments.

Statement of need

On-device AI is now common in production mobile software: retrieval-augmented generation assistants, offline speech and vision models, and privacy-preserving personalization all run inference

directly on a user's phone rather than a server. This shift removes the network as a single point of failure but introduces a different, less well-understood set of failure modes that are specific to constrained, heterogeneous consumer hardware. A model that performs correctly in a controlled benchmark can still fail in the field because the host device is thermally throttled, because available memory has been reclaimed by the operating system, because a quantized model's numerical error compounds over a session, or because a corrupted download silently swaps in the wrong model weights. These failures are rarely caught by conventional unit or widget tests, which assume a stable execution environment and well-formed input.

Existing testing practice for mobile AI features tends to fall back on manual, ad hoc device testing or on general-purpose chaos engineering tools that were designed for backend services rather than for the specific resource and numerical constraints of on-device inference. There is no widely adopted, Flutter-native framework that lets a developer declare "run this model under memory pressure, thermal throttling, and quantization drift, and tell me whether it still meets its confidence threshold" as part of an automated test suite. This gap means AI-specific robustness regressions are frequently discovered by end users rather than by CI, which is costly for teams shipping AI features on tight release cycles.

SATE AI addresses this gap by providing a systematic, reusable, and CI-friendly way to stress-test on-device models within the Flutter ecosystem (Google, n.d.b) that the majority of cross-platform mobile AI applications already use. By expressing each failure mode as an independent, composable injector against a common adapter interface, the framework lets developers start with a single fault injector and grow their coverage incrementally, rather than requiring a full testing framework rewrite. This design was informed by direct experience building and shipping offline-first, on-device AI Flutter applications, where the absence of such tooling had previously meant that robustness issues were found only through manual exploratory testing.

State of the field

General-purpose testing tools for Flutter, such as the `flutter_test` and `integration_test` packages, verify widget behavior and application logic but have no built-in vocabulary for AI-specific failure modes such as quantization drift or confidence collapse. Chaos-engineering tools originating in backend and distributed-systems contexts, such as Chaos Monkey (Netflix, Inc. 2012), popularized the idea of deliberately injecting failures to validate system resilience, but they target infrastructure-level faults (killing processes, dropping network connections) rather than the resource and numerical faults specific to on-device inference on a single mobile device. In the deep-learning testing literature, systems such as DeepXplore (Pei et al. 2017) and DeepTest (Tian et al. 2018) introduced systematic testing of neural networks through metrics like neuron coverage and metamorphic input transformations, primarily for vision models running on servers or in simulation; they do not address deployment-time concerns such as device memory pressure, thermal throttling, or model-file corruption, and they are not integrated with a mobile application framework.

On-device inference runtimes themselves, including TensorFlow Lite (Google, n.d.c) and ONNX Runtime (ONNX Runtime developers, n.d.), provide APIs for loading and running quantized models efficiently on constrained hardware, and their documentation describes best practices for quantization and hardware acceleration, but they do not provide a testing harness for validating that an integrating application degrades gracefully when those constraints are violated. SATE AI is positioned to complement these runtimes rather than replace them: it treats a `TFLiteAdapter` or `OnnxAdapter`-wrapped model as the system under test and applies fault injectors around it, so it

can be adopted incrementally alongside an existing inference pipeline.

What distinguishes SATE AI from the tools above is the combination of (1) fault injectors that specifically target on-device AI failure modes rather than generic infrastructure or generic neural-network correctness, (2) a Flutter/Dart-native implementation that integrates directly into the toolchain most cross-platform mobile AI developers already use, and (3) first-class CI integration through a GitHub composite action and CLI, so stress testing can run automatically on every pull request rather than depending on manual device testing. This combination targets a currently underserved point in the testing landscape: mobile-specific, AI-specific, CI-automatable robustness testing.

Software architecture

SATE AI is organized around three core abstractions: `FaultInjector`, `StressRunner`, and `Report`. A `FaultInjector` implements a single, self-contained failure scenario against a model under test and yields a pass/fail outcome along with diagnostic detail; injectors are independent of one another and can be freely combined. The `StressRunner` orchestrates a stress-test session by applying a list of injectors to a given model adapter and aggregating their outcomes, and the resulting `Report` object exposes the aggregate pass/fail status, individual failures, and export methods for JSON, Markdown, and CSV.

Models under test are wrapped behind a common adapter interface so that the fault injectors are agnostic to the underlying inference runtime. Three adapters ship with the library: `MockAdapter`, a pure-Dart adapter with no native dependencies, intended for testing the framework itself and for demonstrating usage without a real model file; `OnnxAdapter`, which wraps ONNX Runtime for models exported in the ONNX format; and `TFLiteAdapter`, which wraps TensorFlow Lite for `.tflite` models. This adapter pattern is the primary extension point of the framework: supporting a new inference runtime requires implementing the adapter interface rather than modifying any fault injector.

The seven bundled fault injectors cover distinct, empirically motivated failure modes. `MemoryPressureInjector` simulates out-of-memory conditions by constraining available memory during inference. `MalformedInputInjector` exercises empty, oversized, and binary-garbage inputs. `QuantizationDriftInjector` simulates the gradual numerical precision loss associated with quantized model weights. `ThermalThrottleInjector` simulates CPU thermal throttling, a common condition on sustained mobile inference workloads. `LatencyInjector` simulates progressively increasing inference latency. `ModelSwapInjector` simulates model file corruption or an unintended model substitution. `ConfidenceThresholdInjector` validates that a model's output confidence remains above a configurable threshold under the other injected conditions. Because injectors implement a shared interface, third-party injectors can be authored and composed alongside the built-in set without modifying the core library.

Implementation

SATE AI is implemented in Dart (Google, n.d.a) and distributed as a Flutter/Dart (Google, n.d.b) package on pub.dev. The core library, adapters, and injectors are validated by 164 automated unit tests covering all core modules, and the package currently holds pub.dev's maximum static-analysis and documentation quality score of 160/160. Nine releases (v0.1.0 through v0.7.0) have

been published to date, reflecting incremental hardening of the injector set, adapter interface, and reporting layer based on real-world use.

Beyond the core Dart library, SATE AI provides three interfaces for running and consuming stress tests. A browser-based dashboard renders `Report` output for visual inspection, with a dark-mode theme and a mobile-responsive layout, and supports exporting results as JSON, Markdown, or CSV for sharing or archival. A command-line interface allows stress tests to be invoked outside of a Dart test runner, for example `sate_ai stress --model model.gguf --injectors memoryPressure`, which is useful for ad hoc local runs against a specific model artifact. Finally, a GitHub composite action packages the CLI for continuous integration, allowing a repository to fail a pull request automatically when a bundled or newly trained model regresses against its stress-test baseline, in the same way conventional unit tests already gate merges.

Example usage

The snippet below shows a minimal stress test using the pure-Dart `MockAdapter` and two of the bundled fault injectors. The resulting `Report` exposes a boolean `passed` flag and, on failure, a list of individual `failures` plus a Markdown rendering suitable for posting as CI output or a pull-request comment.

```dart
import 'package:sate_ai/sate_ai.dart';

Future<void> main() async {
  final model = MockAdapter(modelId: 'my-model');
  final report = await SateAI.stress(
    model: model,
    injectors: [
      MemoryPressureInjector(limitMb: 100),
      MalformedInputInjector(),
    ],
  );

  if (report.passed) {
    print('PASS: Model passed stress test.');
  } else {
    print('FAIL: Model failed: ${report.failures.length} failures.');
    print(report.toMarkdown());
  }
}
```

Swapping `MockAdapter` for `OnnxAdapter` or `TFLiteAdapter` runs the identical injector list against a real ONNX or TensorFlow Lite model without any other code changes, and the same call can be driven from the CLI or from the GitHub composite action to make stress testing part of a project's regular CI pipeline.

Acknowledgements

The author thanks contributors to the SATE AI repository and the open-source Flutter and Dart communities whose tooling this project builds upon.

References

Google. n.d.a. “Dart: A Client-Optimized Language for Fast Apps on Any Platform.” https://dart.dev.

———. n.d.b. “Flutter: Build Apps for Any Screen.” https://flutter.dev.

———. n.d.c. “TensorFlow Lite: On-Device Machine Learning.” https://www.tensorflow.org/lite.

Netflix, Inc. 2012. “Chaos Monkey.” https://github.com/Netflix/chaosmonkey.

ONNX Runtime developers. n.d. “ONNX Runtime.” https://onnxruntime.ai.

Pei, Kexin, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. “DeepXplore: Automated Whitebox Testing of Deep Learning Systems.” In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 1–18. https://doi.org/10.1145/3132747.3132785.

Tian, Yuchi, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. “DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars.” In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 303–14. https://doi.org/10.1145/3180155.3180220.