The Journal of Open Source Software

Bloch: a strongly typed, hardware-agnostic, hybrid quantum programming language

Akshay Pal [1]

1 Independent Researcher, United Kingdom

DOI: 10.21105/joss.09625

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @HectorMozo3110
- @joshiCoding

Submitted: 23 November 2025
Published: 29 January 2026

Summary

Bloch is a modern quantum programming language and interpreter designed to feel familiar to systems developers while remaining hardware-agnostic. The project combines a statically typed surface language, a semantic analyser that guards against classical/quantum misuse before execution, and an interpreter that emits OpenQASM traces and aggregates multi-shot measurement results. Bloch supports classical control flow, deterministic resource management, and first-class annotations such as `@quantum` (to delineate quantum code) and `@tracked` (to stream measurements into probability tables). The language is implemented in C++20, ships with a comprehensive test suite, and is licensed under Apache-2.0 to encourage adoption in research and industrial workflows. By pairing static semantics with vendor-neutral QASM emission, Bloch targets the gap between dynamically typed Python-first SDKs (e.g., Qiskit and Cirq (Cirq Developers, 2023; Qiskit contributors, 2023)) and vendor-specific languages while keeping a single-binary, non-Python toolchain.

Statement of Need

Quantum practitioners frequently prototype algorithms in Python-first SDKs like Qiskit (Qiskit contributors, 2023) or Cirq (Cirq Developers, 2023), where dynamic typing and runtime-only feedback can make it difficult to catch logic errors prior to simulation or hardware execution. Researchers building higher-level language abstractions (e.g., Silq (Bichsel et al., 2020) and Quipper (Green et al., 2013)) typically need to choose between tightly coupled vendor ecosystems or experimenting with new type systems without a reference interpreter capable of emitting standard assembly formats (OpenQASM 2 (Cross et al., 2017)). Practitioner-friendly hybrids such as PennyLane (PennyLane Developers, 2024) and Microsoft Q# (Microsoft Quantum, 2024) also trade stronger typing or portability for deeper vendor/framework integration. Bloch fills this gap by delivering a self-contained toolchain that is explicitly scoped to the following objectives:

- enforces a compact, explicit type system over both classical and quantum data so that invalid measurements, illegal `@quantum` return types, and improper qubit mutations are rejected at compile time;
- produces OpenQASM 2 output by default, which enables researchers to feed Bloch programs into downstream simulators or device backends without rewriting;
- exposes deterministic multi-shot execution with aggregated statistics so that algorithm designers can validate probabilistic behaviour before running on expensive quantum hardware; and
- stays outside heavyweight Python packaging so that experiments can be reproduced from a single CLI binary rather than a coupled SDK stack.

By targeting developers who are comfortable with systems languages but need a high-level quantum DSL, Bloch reduces the friction between research sketches and reproducible experiments.

Design and Implementation

Bloch's architecture is intentionally modular:

- **Front-end.** A hand-written lexer and Pratt-style parser (under `src/bloch/lexer` and `src/bloch/parser`) build an abstract syntax tree (AST) that covers declarations, control flow constructs, quantum annotations, and array expressions. The AST is shared by the analyser, interpreter, and tests.
- **Static semantics.** The analyser (`src/bloch/semantics`) maintains a scoped symbol table and a compact `ValueType` universe to ensure that only valid combinations of classical and quantum operations progress to execution. It also enforces Bloch-specific rules, such as restricting `@quantum` functions to `bit` or `void` return values and rejecting `@tracked` annotations on unsupported types.
- **Runtime & simulator.** The interpreter (`src/bloch/runtime`) evaluates the AST, orchestrates an ideal statevector simulator, and records measurement outcomes per tracked symbol. Every run emits an OpenQASM trace, and the CLI (`src/main.cpp`) can execute programs for `N` shots, aggregate measurement counts, and emit tabulated probability estimates alongside the generated QASM.
- **Feature staging.** The lightweight feature-flag registry (`src/bloch/feature_flags.hpp`) gives maintainers a way to land experimental language constructs (e.g., an upcoming class system) without destabilising the default build.

The following Bloch kernel produces the interaction shown in Figure 1 when run for a single shot:

```
@quantum function main() -> void {
    qubit q;
    h(q);
    bit r = measure(q);
    reset(q);
}
```

Figure 1 illustrates the runtime interaction between a Bloch program and the ideal statevector simulator for a single kernel, showing gate application, measurement, classical result return, and qubit reset:

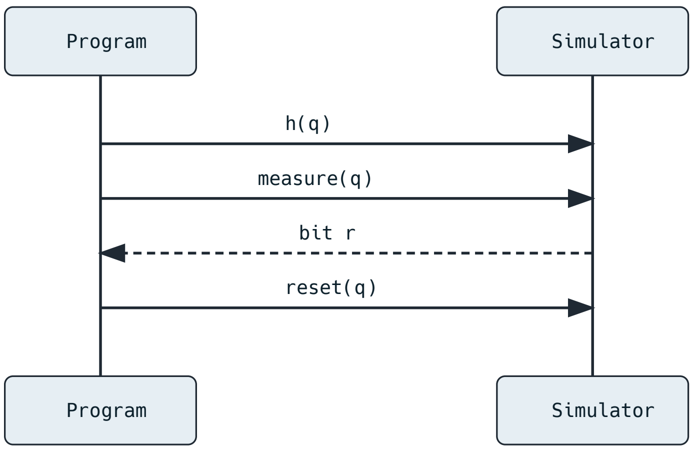

Figure 1: Figure 1: Program-simulator message flow for a Bloch kernel

The project distributes an installable binary and is validated on Linux, macOS, and Windows. Users interact with the CLI (`bloch <file.bloch>`) which offers `--emit-qasm`, `--shots=N`, and `--echo` parameters for reproducible experiments.

Comparative Analysis and Performance

Bloch targets a different point in the design space than popular hybrid SDKs:

Capability	Bloch	PennyLane (PennyLane Developers, 2024)	Q# (Microsoft Quantum, 2024)	Cirq (Cirq Developers, 2023)
Implementation language	C++20 CLI, single binary	Python library with plugin backends	.NET language and runtime	Python library
Type system	Static, quantum-aware	Dynamic (Python)	Static	Dynamic (Python)
Default artefact/output	OpenQASM 2 trace	Backend-specific (optionally QASM)	QIR/targeted executables	Circuit objects (optionally QASM)
Hardware/vendor coupling	Hardware-agnostic via QASM	Plugin-dependent	Azure-centric toolchain	Google-centric, simulators available
Built-in multi-shot aggregation	Yes (`@tracked`, CLI `--shots`)	Backend-provided	Backend-provided	Backend-provided

Lightweight performance baselines from the bundled examples (captured with the ideal simulator and 1024-shot runs) are:

Algorithm (1024 shots)	Bloch execution time (3 s.f.)
Hadamard gate on single qubit	0.006 s
Preparing and measuring a Bell state	0.018 s
Grover search (N = 4)	0.055 s

These numbers come from the average runtime across ten runs for example algorithms provided in the `examples` folder. It illustrates that the native C++ interpreter has negligible startup cost compared to Python-first stacks.

Quality Control

Bloch ships with unit and integration tests implemented with the project's minimal test harness (`tests/test_framework.hpp`). The test suite covers the entire pipeline:

- lexical analysis and token categorisation (`tests/test_lexer.cpp`),
- parser shape and AST formation (`tests/test_parser.cpp`),
- static semantics (e.g., scope rules, `@quantum` return constraints, `final` assignments) in `tests/test_semantics.cpp`,
- runtime behaviour such as OpenQASM emission, measurement persistence inside loops, `@tracked` aggregation, and echo handling in `tests/test_runtime.cpp`, and
- integration smoke tests that exercise representative Bloch programs end to end (`tests/test_integration.cpp`).

Continuous integration executes `ctest` on every pull request, while developers can repeat the same workflow locally via the commands documented in `README.md`. Coverage touches the lexer, parser, semantic analyser, and runtime, providing confidence that regressions in the type system, simulator, or CLI are caught early.

Use Cases

Bloch aims to reduce the distance between whiteboard circuits and evaluable experiments. Current use cases include:

1. **Educational demos.** The Bell-state example (`examples/02_bell_state.bloch`) showcases hardware-agnostic entanglement with deterministic statistics reporting, making it suitable for classroom explanations or live coding.
2. **Algorithm sketching.** Researchers can iteratively design algorithms that mix classical control flow with quantum kernels, then export the emitted OpenQASM for downstream tooling without reimplementation. The `examples/04_grover_search.bloch` program demonstrates marked-item recovery via amplitude amplification without any Python runtime dependencies.
3. **Runtime experimentation.** The `@tracked` facility and `--shots` flag make it straightforward to explore noise-free distributions and verify that optimisations preserve measured behaviour before porting kernels to other stacks. These facilities extend to broader application areas such as oracle-based search, small-n optimisation experiments, and amplitude estimation sketches that can be re-targeted by swapping the QASM consumer.

Availability

Bloch is openly developed at https://github.com/bloch-labs/bloch under the Apache-2.0 license. The repository bundles setup instructions, contribution guidelines, and a list of feature flags so that new contributors can propose language extensions while maintaining release stability. Pre-built binaries are available but not required; the CMake toolchain builds the CLI across major platforms, and the project's documentation hub (https://docs.bloch-labs.com) provides user guides and API notes.

Acknowledgements

I thank the Bloch open source community for issue reports, early design feedback, and preview testing, as well as the maintainers of the upstream projects cited in this paper whose work makes Bloch interoperability possible.

References

Bichsel, B., Baader, M., Gehr, T., & Vechev, M. (2020). Silq: A high-level quantum language with safe uncomputation. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 286–300. https://doi.org/10.1145/3385412.3386010

Cirq Developers. (2023). *Cirq*. https://doi.org/10.5281/zenodo.4062499

Cross, A. W., Bishop, L. S., Smolin, J. A., & Gambetta, J. M. (2017). *Open quantum assembly language*. https://arxiv.org/abs/1707.03429

Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., & Valiron, B. (2013). Quipper: A scalable quantum programming language. *Proceedings of the 34th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, 333–342. https://doi.org/10.1145/2491956.2462177

Microsoft Quantum. (2024). *The q# programming language*. https://learn.microsoft.com/azure/quantum/user-guide/programming/qsharp/

PennyLane Developers. (2024). *PennyLane: Quantum machine learning and differentiable programming*. https://pennylane.ai

Qiskit contributors. (2023). *Qiskit: An open-source framework for quantum computing*. https://doi.org/10.5281/zenodo.2573505