Barista: Synthesizing Typestate Specifications with LLM Agents

Catarina Gamboa
Carnegie Mellon University
Pittsburgh, USA
LASIGE, University of Lisbon
Lisbon, Portugal
cgamboa@andrew.cmu.edu

Paulo Canelas
Carnegie Mellon University
Pittsburgh, USA
LASIGE, University of Lisbon
Lisbon, Portugal
pasantos@andrew.cmu.edu

Ricardo Costa
LASIGE, University of Lisbon
Lisbon, Portugal
rimcosta@lasige.di.fc.ul.pt

Márcio Caetano
LASIGE, University of Lisbon
Lisbon, Portugal
mmilisse@lasige.di.fc.ul.pt

Jonathan Aldrich
Carnegie Mellon University
Pittsburgh, USA
jonathan.aldrich@cs.cmu.edu

Alcides Fonseca
LASIGE, University of Lisbon
Lisbon, Portugal
amfonseca@ciencias.ulisboa.pt

Abstract

Object-oriented classes encode implicit protocols that clients must follow, such as connecting a socket before sending data or opening a stream before reading. Mainstream programming languages like Java cannot validate the correct use of these protocols, but more advanced type systems, such as typestate systems, can detect these violations at compile time. However, these systems require a formalization of these implicit protocols. To overcome this limitation, we propose to extract object protocols from documentation using a combination of LLMs and test generation. We developed Barista, an agentic workflow that transforms class documentation into verifiable LiquidJava specifications through DFA-guided protocol extraction, test synthesis, and verification-driven refinement. We evaluated our approach on 35 Java classes with expert-written ground truth, achieving 87.0% precision and 68.3% recall, with 75% of specifications usable directly or with minor edits. Barista bootstraps these specifications for well-documented libraries, reducing the manual annotation burden and enabling compile-time protocol safety without specialized expertise.

CCS Concepts

• **Software and its engineering** → **Specification languages**; • **Computing methodologies** → Information extraction.

Keywords

Liquid Types, Object Protocol, Code Generation, Agents

ACM Reference Format:
Catarina Gamboa, Paulo Canelas, Ricardo Costa, Márcio Caetano, Jonathan Aldrich, and Alcides Fonseca. 2026. Barista: Synthesizing Typestate Specifications with LLM Agents. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3834427

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834427

1 Introduction

Classes in object-oriented programming often define implicit protocols that clients should follow to properly interact with their APIs [10, 48]. For example, a `Socket` must be connected before sending data, a `FileInputStream` needs to be open before reading, and once closed, neither can perform these operations again. When developers violate these protocols, the resulting bugs are difficult to detect, as they may only manifest under specific runtime conditions that unit tests fail to exercise [3, 35].

Typestate systems offer a principled solution to this problem by encoding valid method sequences as type-level state machines that the compiler verifies statically, before the code executes [2, 45]. Researchers have developed several tools that bring typestate checking to object-oriented languages, including Plaid [1], Plural [9], Mungo [27], JaTyC [32], and LiquidJava [21], demonstrating that protocol constraints can be expressed formally and verified statically against client code.

However, like formal verification tools more broadly [34], adoption remains limited despite their potential to catch critical bugs early. Developers *"rarely go out of their way"* [23] to write formal specifications that translate their understanding of code behavior into verifiable properties [22]. Yet this knowledge exists, as developers encode protocol constraints in Javadoc comments and API documentation [28, 47, 59]. For example, the Javadoc for `Socket` explicitly states that `getInputStream()` throws an exception if "the socket is not connected." The gap lies not in missing knowledge, but in translating this informal documentation into formal specifications that can be used to statically detect property violations.

Previous work on mining object protocols [26, 53, 55] depends on existing client code or execution traces, which are not always available and may not cover all relevant usage patterns. Recently, Large Language Models (LLMs) show promise in automating verification-related tasks when combined with static analysis [36, 50], and researchers have used them to generate method-level pre- and post-conditions from documentation [17, 38]. However, these approaches do not reason about abstract object states or valid method sequences, which are essential for encoding class-level protocols. To the best of our knowledge, no existing approach combines LLM-based specification generation with the ability to capture both method-level constraints and class-level protocols that span an object's lifetime.

To this end, we present Barista, an agentic workflow that transforms class documentation into verifiable LiquidJava specifications. We target LiquidJava because it combines typestate annotations with refinement types [39] for more expressive protocols that have been shown to be effective in detecting protocol violations. Additionally, developers using LiquidJava find more protocol bugs in less time [21]. Barista brings these benefits to any developer, delivering compile-time verification without the annotation burden. Our approach uses specialized LLM agents to extract the class protocol as a deterministic finite automaton (DFA), generate test cases for valid and invalid usage patterns, produce refinement annotations, and iteratively refine specifications based on verification feedback.

We make the following contributions:

- **An agentic workflow** transforming class documentation into verifiable LiquidJava specifications through DFA generation, test synthesis, and verification-driven refinement, enabling fully automated specification synthesis (Section 4);
- **A benchmark of 35 Java classes** with expert-written specifications covering diverse domains and protocol types from prior empirical work [8], released to enable evaluation of future synthesis approaches (Section 5.2);
- **Empirical evidence on synthesis effectiveness.** We show that generated specifications achieve 87.0% precision, 68.3% recall, and 76.5% F1 against expert test suites, with 75% usable directly or with minor edits (Section 5.4);
- **An ablation study on design choices.** We isolate the impact of each pipeline component and show that DFA-guided generation significantly improves precision with a large effect size, while test-driven validation significantly improves recall with a small effect size (Section 5.5).

The remainder of the paper presents the background and motivational example (Section 2), related work (Section 3), and details of our approach (Section 4), followed by an evaluation of the generated specifications (Section 5), a discussion of limitations and future work (Section 6), and concluding remarks (Section 7).

2 Background and Motivational Example

LiquidJava [21] extends Java's type system with refinement types [39], enabling developers to express both object protocols and constraints on method arguments and return values in a single framework. Developers write logical predicates in a decidable logic, and the type checker validates them at compile time using an SMT solver [13]. Uninterpreted functions model abstract states, and method annotations specify the "from" and "to" states of each transition.

Let us consider Java's `Socket` class, with the documentation:

> `public Socket()` Creates an unconnected Socket.
> `public Socket(InetAddress address, int port)`
> Creates a stream socket and connects it to the specified port number at the specified IP address. [...]
> *IllegalArgumentException* - if the port parameter is outside the specified range of valid port values, which is between 0 and 65535, inclusive.
>
> - javadoc of `java.net.Socket`

The documentation hints that the two constructors produce different initial states: unconnected and connected. The complete Javadoc reveals seven states in total (Listing 1). We capture state transitions by annotating each method with `@StateRefinement(from = "state1", to = "state2")`. Additionally, we encode argument constraints such as the port restriction using `@Refinement("port >= 0 && port <= 65535")`.

```
1  import liquidjava.specification.*;
2  import java.net.InetAddress;
3  import java.net.SocketAddress;
4
5  @StateSet({"unconnected", "bound", "connected", "inputshutdown"
       , "outputshutdown", "bothshutdown", "closed"})
6  @RefinementAlias("ValidPort(int p) {p >= 0 && p <= 65535}")
7  @ExternalRefinementsFor("java.net.Socket")
8  public interface SocketRefinements {
9      // Constructors
10     @StateRefinement(to = "unconnected(this)")
11     public void Socket();
12
13     @StateRefinement(to = "connected(this)")
14     public void Socket(InetAddress address,
15                        @Refinement("ValidPort(_)") int port);
16
17     // Connection methods
18     @StateRefinement(from = "unconnected(this)",
19                      to = "bound(this)")
20     public void bind(SocketAddress bindpoint);
21
22     @StateRefinement(from = "bound(this)",
23                      to = "connected(this)")
24     public void connect(SocketAddress endpoint,
25                         @Refinement("_ >= 0") int timeout);
26
27     @StateRefinement(from = "connected(this)")
28     public void sendUrgentData(int data);
29
30     // Shutdown methods, shutdownOutput is similar
31     @StateRefinement(from = "connected(this)",
32                      to = "inputshutdown(this)")
33     @StateRefinement(from = "outputshutdown(this)",
34                      to = "bothshutdown(this)")
35     public void shutdownInput();
36
37     @StateRefinement(to = "closed(this)")
38     public void close();
39 }
```

Listing 1: LiquidJava annotations required for the `Socket` class with typestate and argument refinements.

Beyond the features present in the `Socket` example, LiquidJava supports state transitions that depend on argument values. For instance, `@StateRefinement(to="multipleSelectionsAllowed ? multiple(this) : single(this)")` is used in the `ChoiceCallback` class[1] where the ternary operator routes the object to different states based on the value of the `multipleSelectionsAllowed` argument.

Furthermore, LiquidJava supports multiple state sets to capture orthogonal properties within the same class. For instance, the `ImageWriteParam` class[2] tracks both tiling (on, set, off) and compression (on, set or off). Rather than creating nine states for all combinations, we define two state sets, each with its own on, set and off states, and methods that transition the object within each set independently.

[1] javax.security.auth.callback.ChoiceCallback
[2] javax.imageio.ImageWriteParam

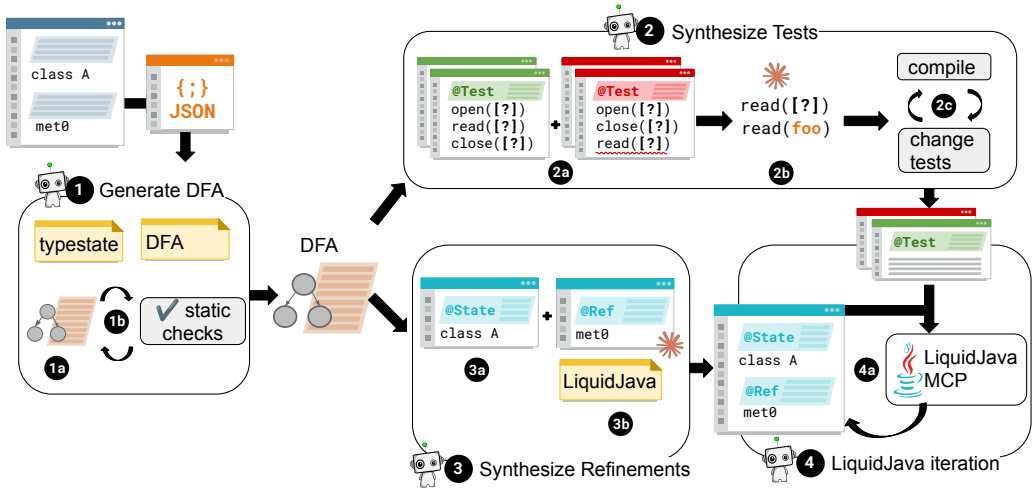

Figure 1: Agentic workflow to generate refinements. The approach starts with class documentation parsed into JSON format. Step 1 generates a DFA: the agent interprets documentation into typestate (1a), then validates the state machine iteratively until it passes all checks (1b). The DFA is used for two parallel synthesis tasks: tests (2) and refinements (3). In step 2, we enumerate passing and failing test templates (2a), complete them (2b), and iterate until they compile (2c). In step 3, we generate state refinements from the DFA (3a) and use the LLM to generate parameter and return constraints (3b). Finally, in step 4, we verify that the LiquidJava annotations correctly classify the test suite (4a), iteratively refining specifications if any test fails.

While expressive, writing these annotations manually requires significant effort and expertise. Our goal is to automatically generate these specifications from class documentation using LLM agents, enabling static verification of protocol violations without manual annotation effort.

3 Related Work

Protocol specifications can be obtained from different sources, each with distinct limitations. Whaley et al. [55] and Daikon [18] extract finite-state models and invariants, respectively, from instrumented code traces, while Weimer and Necula [53] mine temporal safety specifications from program traces, and Kechagia et al. [26] detect API misuses via exception propagation in search-based testing. These trace-based approaches are inherently limited to the paths exercised by tests, requiring extensive test suites with no guarantee of covering the complete protocol. Amann et al. [3] confirm these limitations, reporting that state-of-the-art API misuse detectors suffer from low precision and recall in realistic settings. Template-based approaches that target source code also face challenges. Houdini [20] primarily generates annotations from heuristic templates but supports only simple logical expressions.

Documentation offers a complementary source of protocol knowledge, and recent work leverages LLMs to extract it. LLMs can generate pre- and postconditions from documentation [17, 38], and subsequent work combines LLMs with verification-driven refinement to iteratively improve specifications [30, 54]. Although these approaches generate function-level specifications effectively, they do not encode valid method sequences across an object's lifetime. In particular, recent specification-synthesis tools such as SpecGen [30] and GPTAid [29] target parameter- and function-level rules, whereas Barista captures class-level protocols that span an object's lifetime.

At the class level, ClassInvGen [46] generates class invariants and test inputs for C++ classes, but invariants capture properties that hold at stable points rather than state transitions that define object protocols. Yet LLMs have demonstrated the capability to extract protocol state machines from implementation source code [52, 58], suggesting this capability can be directed toward generating verifiable specifications from documentation. These efforts continue a long line of state-machine inference [24] that learns behavioral models from observed executions rather than from documentation.

Together, these approaches leave a gap as none derive verifiable, class-level protocol specifications from documentation alone. We address it with Barista, transforming class documentation into verifiable LiquidJava specifications that encode both state transitions and method-level constraints.

4 Approach

Figure 1 presents our workflow for transforming Java class documentation into verifiable specifications that capture both class protocols and method-level constraints. The workflow focuses on four steps: the first extracts a state machine representing the class typestate, the second synthesizes passing and failing tests from this state machine, the third produces LiquidJava annotations with typestate and argument refinements, and the fourth iteratively refines specifications based on test results. In our multi-agent implementation, each step is executed by a specialized agent that interacts with the others, producing annotated classes that enable compile-time detection of protocol violations without manual specification

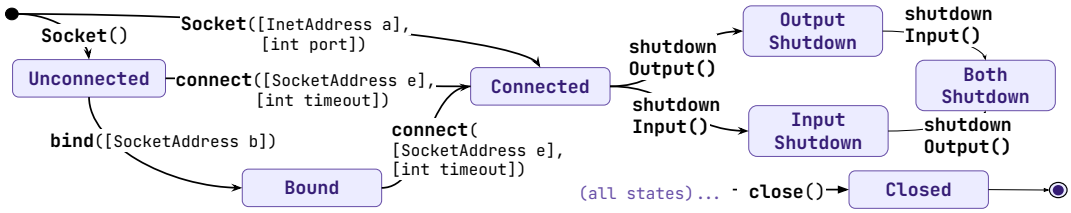

Figure 2: Simplified version of the DFA generated by Agent 1 for the `Socket` class. There are seven states present and methods that change the object state.

effort. Since this approach extracts protocols only from documentation, we rely on the precondition that the target class must be well documented.

The design of these agents is drawn from other successful examples. We integrate verification tools in-the-loop to leverage error message feedback for guided generation [30, 54], and we provide task-relevant context to each agent, such as DFA construction rules and LiquidJava syntax documentation [11, 51]. We also decompose the workflow into specialized agents that focus on distinct subtasks, following the philosophy that mimicking expert behavior improves generation quality [57].

4.1 DFA Generation

Typestate systems traditionally represent class protocols as deterministic finite automata (DFA), where states capture object configurations and transitions encode valid method sequences [1, 45]. The first step of our workflow is to generate a DFA that captures the class protocol from its documentation.

The first agent receives the class documentation as structured JSON, parsed from the available HTML documentation, containing the Javadoc comments, class description, method signatures, and field descriptions. From this input, the agent generates a state machine that captures the class protocol and saves it as a Mermaid diagram.[3] To guide this generation, we provide documentation explaining typestate and its usage, drawing on prior work [8] that includes common protocol categories and a classic `File` example with `open` and `closed` states.

After generating an initial DFA, the agent validates it using a custom Model Context Protocol (MCP) tool that verifies all public methods are present, all states are reachable, and the automaton is deterministic. If any check fails, the agent iterates using the tool's feedback until the DFA passes all validation criteria. Figure 2 shows a simplified DFA generated for the `Socket` class, retaining expected arguments to differentiate transitions based on parameters.

The generated DFA serves two purposes: we derive class states and method transitions from it to annotate typestate constraints, and we synthesize test cases by generating method call sequences that correspond to valid and invalid paths, verifying that specifications correctly detect protocol violations.

4.2 Refinements Synthesis

The Mermaid DFA provides a structured textual representation of the class protocol that, using a custom MCP tool, we mechanically translate into LiquidJava by mapping DFA states to `@StateSet` and transitions to `@StateRefinement` annotations on methods.

The refinement synthesis agent calls this MCP tool to generate the skeleton, adding the necessary imports and argument refinements. To generate these constraints, the agent consults the JSON documentation along with a guide describing how liquid types constrain inputs and outputs. From this information, the agent creates predicates for primitive types (e.g., `int`, `long`) and identifies repeated patterns to generate reusable refinement aliases. The final annotated class contains all necessary imports, typestate refinements, and argument constraints.

With both the DFA and the annotated class in place, we need to verify that the annotations correctly capture the class protocol. Therefore, we need test cases that represent valid and invalid usage patterns of the class.

4.3 Test Synthesis

By iterating through the different valid (and invalid) paths in the DFA, we can generate chains of method calls that act as valid (and invalid) test cases. Since test cases will be statically verified using LiquidJava, we only require tests to be syntactically correct and typecheck according to Java's type system.

Overall, our test generation approach follows a three-step approach. First, valid and invalid paths in the DFA are encoded as method sequences, creating test templates. A custom MCP tool performs that task by traversing the DFA with a depth-first search with limits on maximum depth, consecutive method calls, and self-loops, along with a minimum path length to ensure sufficiently complex examples. From this initial set of paths, we filter duplicates and rank the remaining tests to maximize coverage of state changes and distinct states visited. Then, the agent takes these test templates, fills the placeholders, and adds the necessary imports. For instance, the agent can have a valid usage pattern for the `Socket` class `Socket s = new Socket([ InetAddress addr, int port]); s.close();`, where the bracketed arguments indicate placeholders that the agent must fill in. Finally, the agent tries to compile the tests and iteratively refines them until compilation succeeds, producing a complete test suite.

4.4 LiquidJava Iteration

The final step of the pipeline verifies that the generated specifications correctly capture the class protocol by checking whether the result of each test agrees with the result of the LiquidJava verifier. When a test fails the verification or produces an unexpected result,

[3] https://mermaid.js.org/intro/

Table 1: Dataset of 35 Java classes with typestate protocols, grouped by domain. Bookmarks point to JDK17 Javadoc documentation. For each domain, we report the number of classes of that domain and the ranges of the number of methods across classes.

Domain	Classes	#	Sizes
Graphics	AbstractUndoableEdit, Clipboard, DefaultMutableTreeNode, DragSourceContext, DropTarget, FlatteningPathIterator, ImageReadParam, ImageWriteParam, PopupMenu, UndoManager, Window	11	5–89
I/O	BufferedInputStream, BufferedReader, DeflaterOutputStream, PipedOutputStream, PipedWriter, ZipFile, ZipOutputStream	7	7–14
System	MLet, RequiredModelMBean, ThreadGroup, Throwable, UUID, Vector	6	15–53
Network	DatagramSocket, RMIConnector, ServerSocket, Socket	4	13–54
Security	ChoiceCallback, KerberosKey, KerberosTicket, LoginContext	4	8–26
Data	SQLException, TransformerException, XMLFilterImpl	3	13–35
	Total	**35**	**5–89**

the agent refines the annotations using the error messages until all tests are correctly classified or a maximum number of iterations is reached. This process ensures that the generated specifications are syntactically correct and faithfully capture the class protocol semantics as expressed in the test suite.

4.5 Implementation Details

We designed this workflow in a new tool, Barista, combining LiquidJava with LLM agents to automate the generation of typestate specifications from class documentation. We implemented Barista using Claude Code [5] with the Sonnet 4.6 model, which ranked as the top model for software engineering tasks on the SWE-Bench leaderboard [25] at the time of writing. Each agent receives a task description as a prompt along with a specification of available tools, and all static tools follow the Model Context Protocol (MCP), allowing agents to invoke them as needed. We provide documents with specialized information, such as typestate definitions and LiquidJava syntax guides, as skills [6], which agents access whenever they require relevant context. To orchestrate the complete workflow, we use Claude's Agent SDK via its Python API [4] and implement a command-line tool to run the entire process for a given class. The full implementation is available in our replication package [7]. For reproducibility, this package includes the full agent prompts, the skill documents, the MCP tool implementations, and the exact DFA path-enumeration parameters used in test synthesis. This implementation was designed to be modular and show each step of the pipeline separately, but the same decomposition could be implemented in a single agent or a different SDK.

5 Evaluation

To evaluate our agentic approach, we compare automatically generated specifications against expert-written ones across a diverse set of Java classes known to exhibit typestate protocols. For our evaluation, we focus on two research questions:

RQ1 To what extent are the generated specifications effective in correctly capturing object protocols?

RQ1.1 To what extent do the generated specifications correctly distinguish between **valid and invalid protocol** usages in the expert test suite?

RQ1.2 How do automatically generated specifications align with **expert-written specifications**, and what causes divergences?

RQ2 What is the impact of **key design choices** such as DFA-guided generation, test-driven validation, and verification-driven refinement, on specification generation success?

This section describes our dataset selection, the manual annotation process, the automated generation procedure, and the metrics we use for comparison.

5.1 Dataset Selection

We used 35 classes from a prior study on object protocols in Java by Beckman et al. [8] with available online data.[4] We filtered for classes available in JDK17, the version currently supported by LiquidJava, that are public classes with public constructors, and for which a concrete implementation exists. The final list includes classes with different protocol types, domains, and sizes, as presented in Table 1. We included classes from all different *protocol types* identified in the original study, such as Initialization, Liveness Check, Deactivation Check, Boundary Check, Redundant Request, and Marker State. We covered multiple domains, including I/O, Networking, Graphics, Security, System, and Data. And, finally, classes of varying *size*, ranging from 5 to 89 public methods.

5.2 Manual Expert Annotation

To establish a ground truth for comparison, two authors independently annotated each class following a standardized protocol, adopting a dual-annotator approach recommended for establishing reliable ground truth in empirical software engineering studies [12, 43]. We developed an annotation guide outlining the expected outputs and decision criteria. For each class, annotators produced a LiquidJava specification file with the annotations, along with an optional DFA diagram and tests with example usages. Beyond the specification itself, annotators recorded metadata to support later analysis: the time spent on the annotation, the number of states and transitions identified, the number of methods annotated (and how many included argument refinements), and any refinement aliases created. Annotators also documented their decision

[4] https://www.nelsbeckman.com/research/esopw/

rationale for ambiguous cases such as unclear state transitions, methods callable in multiple states, or uncertain self-loops, and noted any tool limitations where properties could not be expressed in LiquidJava. This documentation creates an audit trail that supports traceability and makes our interpretive choices transparent and reproducible [40]. For this process, annotators could read the documentation, search the web for relevant information, look at source code, and even execute the code, but using AI tools was not allowed to reduce the risk of biasing the ground truth with the same sources used for generation. These details were all included in the annotation guidelines.

After independent annotation, we held an alignment meeting where each author pair met to reconcile their specifications for each class. They compared their DFAs, discussed disagreements, and agreed on a single specification recording the final decisions made. This reconciliation process ensures that our ground truth reflects careful consideration of ambiguous documentation rather than a single annotator's interpretation, addressing a common threat to validity in specification-related studies where expert judgment plays a significant role.

With the annotated classes, we then extended the test suite by creating more valid and invalid tests for each class according to the expert annotations. We computed the coverage of these tests against the expert annotations by gathering *state coverage*, to check if every state in the state machine is visited at least once by a correct test; *transition coverage*, to see if every edge is exercised at least once in the correct tests; and *forbidden-pair coverage*, to see if every invalid pair of (state, method) is tested by at least one incorrect test. The last metric is particularly relevant as it shows the ability of the test suite to capture invalid protocol usages; it was inspired by the concept of forbidden state–action pairs, which identify state–action combinations that may lead to dead-end states [33].

Additionally, we developed a rubric to evaluate the *complexity* of each class's protocol, based on factors such as having branching and conditional transitions, having multiple state sets, or using workarounds due to tool limitations. Two annotators independently ranked the complexity of each class based on these factors and assigned a complexity rating of simple, moderate, or complex, following the same consensus-based approach as for the specifications.

5.3 Evaluation Setup

Cross-Evaluation Protocol. To evaluate the effectiveness of the generated specifications, we used a cross-evaluation protocol where we paired the agentic specifications with the expert test suite. Each test case is labeled according to the confusion matrix (Table 2).

Table 2: Confusion matrix used to label each test case in the cross-evaluation protocol.

	LiquidJava Error	**LiquidJava Passes**
Invalid path	True Positive (TP) *correct detection*	False Negative (FN) *under-restriction*
Valid path	False Positive (FP) *over-restriction*	True Negative (TN) *correct acceptance*

Using these metrics, we compute Precision, Recall, and F1 scores for each class and across the entire dataset. Precision measures the fraction of reported errors that are real, recall measures the fraction of actual violations caught, and F1 provides their harmonic mean. We select these metrics because they separately capture the two failure modes most relevant to static analysis adoption, as false alarms decrease developer trust, and missed bugs leave possible bugs undetected [15, 41]. To account for the non-determinism of the generation process, we ran each class five times.

Ablation Variants. For RQ2, we assess how each component contributes to specification quality by implementing ablation variants and evaluating them using the same cross-evaluation protocol. The *baseline* variant uses few-shot prompting without any pipeline components, as it receives the class documentation and generates specifications directly. The *DFA* variant adds DFA-guided generation with iterative refinement but omits test synthesis. The full *Barista* variant combines all three components: DFA generation, test synthesis, and verification-driven refinement. We computed the same performance metrics for each variant to isolate the contribution of individual components to specification accuracy. Additionally, we compare cost and time required for each variant using the agent logs and the expert-annotated times.

Statistical Tests. To assess whether specification *complexity* predicts the metrics, we compute Spearman rank correlations [44] between the ordinal complexity (simple, moderate, complex) ratings and each metric (Precision, Recall, F1), as it makes no distributional assumptions and is appropriate for ordinal predictors.

For the ablation study, we first pool TP/TN/FP/FN counts across the five runs of each class to obtain a single per-class score, then use the Wilcoxon signed-rank test [56] to compare pipeline variants by pairing these per-class scores across approaches. This non-parametric test is suited to our small sample size ($n \leq 35$ classes) without requiring normality. We report Cliff's δ [31] as an effect-size measure, interpreted as negligible (< 0.147), small (< 0.33), medium (< 0.474), or large (≥ 0.474), and all significance thresholds are set at $*$, $p<0.05$, $**$, $p<0.01$, $***$, $p<0.001$.

Qualitative Analysis. The quantitative metrics above measure *how much* the agentic specifications diverge from expert specifications, but they do not reveal *why* these divergences occur. To answer this question, we conducted a manual qualitative review of all classes with typestate protocols from both the agentic runs and the expert annotations. For each class, one evaluator compared a randomly selected agentic run with the expert specification. We analyzed each specification pair along four dimensions. *State Alignment* captures whether the agent's states match the expert's (exact match, subset, superset, or partial overlap). *Transition Alignment* assesses method transitions (aligned, overly restrictive, overly permissive, or mixed with some transitions overly restrictive and others overly permissive). *Refinement Alignment* evaluates parameter and return refinements (aligned, partial, missing, or not applicable). Finally, *Usability* rates whether developers could make use of the generated specification directly (yes, with minor edits, or no). Beyond these structured dimensions, the evaluator documented in free text what caused each divergence. We then performed open coding [42] on these descriptions, identifying root causes and producing a codebook with definitions and illustrative examples for each pattern.

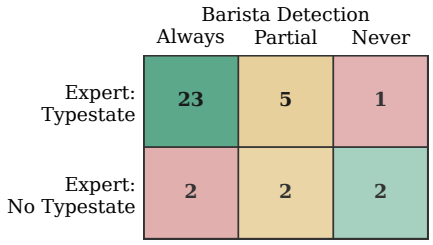

Figure 3: Typestate detections across 35 classes (5 runs each). Green = correct, amber = inconsistent across runs, red = incorrect. Barista detects 28/29 classes in at least one run.

5.4 Specification Effectiveness (RQ1)

This section presents the results of expert annotations (Section 5.4.1), Barista's protocol detection (Section 5.4.2), cross-evaluation with expert tests (Section 5.4.3), and the qualitative analysis of divergences between agentic and expert specifications (Section 5.4.4).

5.4.1 ***Expert Annotations.*** The expert annotators specified 29 of the 35 classes in our dataset. We excluded the remaining six because they lacked a clear protocol, contained constraints beyond the expressiveness of LiquidJava, or consisted primarily of deprecated or protected methods that our annotation guidelines excluded.

During reconciliation, annotators compared their independently created specifications and discussed divergences to reach consensus. Only 10 classes (29%) had exact agreement on the number of states, while 17 (49%) showed divergences ranging from 1 to 6 states, and 2 (6%) were classified as having a protocol by one annotator but not the other. These metrics, however, are just a proxy for the protocol encoding since they focus on the *number of states*, a structural property of the DFA, rather than the behavior it allows. Two DFAs with different state counts can accept exactly the same method sequences, which may happen when one annotator adds redundant states or prefers a flat machine over orthogonal state sets. To analyze these divergences, we looked at the design rationales documented during annotation to see the reasons for these discrepancies. Some annotators included states that could not be verified in LiquidJava (3 classes) or added redundant states (3 classes). Others differed in their preference for flat state machines versus orthogonal state sets (3 classes), missed states that their counterpart identified (3 classes), or disagreed on how deeply to model the protocol (2 classes). For the two classes where annotators disagreed on protocol existence (`KerberosTicket` and `RequiredModelMBean`), experts ultimately agreed to include specifications after determining that both could be meaningfully modeled.

These divergences reflect the inherent ambiguity in interpreting documentation and the subjective nature of deciding what constitutes a protocol state. Since detecting whether a class has a protocol is itself a relevant task, we included all 35 classes in the evaluation of typestate detection, but only the 29 with expert annotations in the cross-evaluation of specifications.

Table 3: Performance metrics for Barista.

	Accuracy	Precision	Recall	F1
Barista	74.8%	87.0%	68.3%	76.5%

5.4.2 ***Protocol Detection.*** Figure 3 shows the typestate detection in Barista, showing that 25 classes match the expected expert protocol detection, with 23 consistently detected as having typestate protocols, and 2 as not having typestate protocols. The remaining 10 classes show less consistent results. From the expected typestate classes, in 5 of them Barista misses the expected protocol in at least one run (*Inconsistently Detected*), and in 1 class it never detects the expected protocol (*Missed*). As for the classes that are not expected to have typestate protocols but Barista detected them (*False Detection*), in 2 of them Barista incorrectly detects a protocol in all runs, and in the remaining 2 it incorrectly finds a protocol at least once.

To understand these discrepancies, we manually examined the agent logs for all divergent runs. The *Missed* case (`Vector` class) reflects actual specification ambiguity, as the expert-defined states serve as a workaround for modeling collection size rather than a clear protocol, and the agent reasonably classified it as a utility class without a lifecycle. The *False Detection* cases stem from the agent modeling runtime-dependent properties as static states. For instance, it created `available`/`unavailable` states for `Clipboard` based on platform-specific restrictions that cannot be determined without information about the execution environment. The *Inconsistently Detected* cases reveal that agents dropped the protocol when they could not express it in the DFA formalism and were overly rigid in their heuristics for recognizing protocols. For example, agents required explicit `IllegalStateException` declarations to recognize a protocol, but these were not always present.

5.4.3 ***Cross-evaluation of Specifications.*** For the 29 classes with expert annotations, we cross-evaluated the Barista-generated specifications against the expert test suite, consisting of 3,262 test files across all classes and runs (1,302 correct tests and 1,960 incorrect tests). Table 3 presents Barista's performance: 74.8% accuracy, 87.0% precision, 68.3% recall, and 76.5% F1 score. The high precision suggests that detected protocol violations are unlikely to be false alarms, while the recall indicates that Barista catches roughly two-thirds of actual violations.

To understand whether class properties influence these metrics, we computed Spearman rank correlations between specification performance and the number of methods, identified states, transitions, and annotated protocol complexity. We found significant correlations only with protocol complexity, where all metrics degrade as complexity increases (Figure 4).

Notably, false positives concentrate in specific classes rather than appearing systemically: 12 of 29 classes (41%) produce zero false positives across all five runs, and just six classes account for 80% of all false positives, all of them moderate or complex protocols. This concentration suggests that specification errors stem from specific, recurring challenges rather than general unreliability. To identify these challenges, we conducted a qualitative analysis comparing a randomly selected agentic run against the expert specification

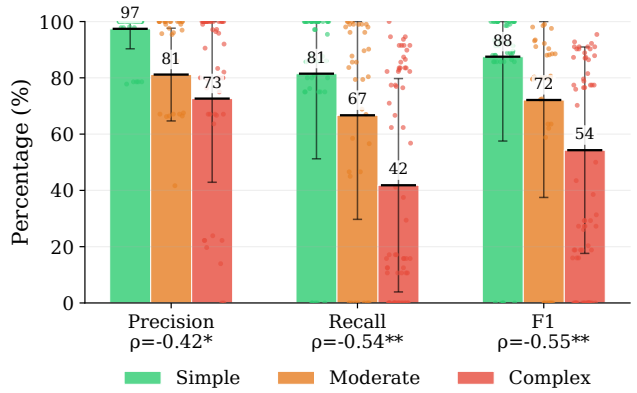

Figure 4: Per-class metric means by protocol complexity, with per-run standard deviation and individual run results. Spearman rank correlations (ρ) between complexity and each metric are shown below the axis. Neither state count, method count, nor number of parameter refinements showed consistent significant correlations. * p<0.05, ** p<0.01.

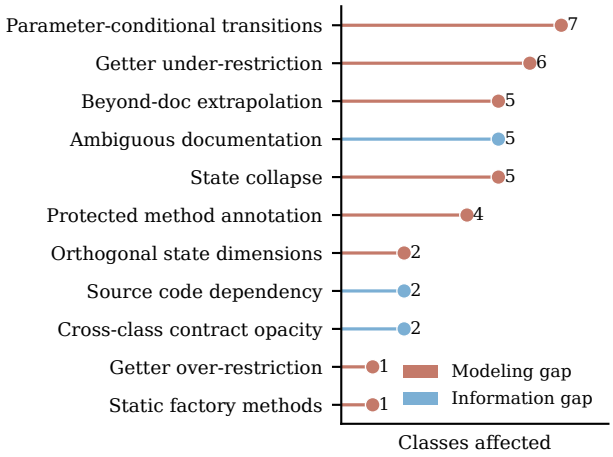

Figure 5: Distribution of root causes for state and transition divergences between agentic and expert specifications.

for each of the 28 typestate classes Barista consistently detects (excluding the *Missed* `Vector` class, Section 5.4.2).

Answer to RQ1.1

Barista produces specifications that correctly classify protocol usages with high precision (87.0%) and moderate recall (68.3%), meaning most flagged violations are real but 30% of actual violations go undetected. Nevertheless, as the complexity of classes increases, the overall agent performance degrades.

5.4.4 ***Qualitative Analysis of Divergences.*** We compared a randomly selected agentic run against the expert specification on four dimensions: state alignment, transition alignment, refinement alignment, and usability. We additionally documented the root causes of divergences, as only 5 of the 28 classes with typestate protocols showed full alignment across all dimensions, having the same states, transitions, and refinements.

For state alignment, 16 out of 28 classes had the exact same states (57%), nearly double the 29% agreement between experts (Section 5.4.1). When divergences occurred, the agent consistently identified fewer states than experts (7/28 subset) rather than identifying more (3/28 superset) or partially overlapping ones (2/28). This conservative bias extends to transitions, as in 10 of 28 classes, the agent permitted methods in states where experts forbade them (under-restricted). For example, in the `BufferedReader` class, the agentic specification allows calling `lines()` in the `Closed` state. As for over-restriction, only 4 of 28 classes had transitions forbidding valid method calls, and 5/28 had a mix of under- and over-restriction. Refinement correctness was the least affected dimension: of the 17 classes where experts defined value-level refinements, 9 (53%) were fully aligned, as these constraints are more mechanical and less dependent on protocol reasoning. However, 2 of 17 classes had missing refinements, one of which stemmed from uninformative verification feedback. In `PipedWriter`, the agent initially combined refinements with state transitions, but when verification failed without providing an informative error message (i.e., "Verification failed with 0 error(s)"), the agent responded by trying several alterations until it removed the refinements altogether. This led to successful verification but a missing refinement in the final specification, illustrating how silent failures can degrade specification quality.

To understand what drives these divergences, we coded the root causes of state and transition misalignments across the 22 of the 23 divergent classes that diverged in state or transition alignment; the remaining divergent class differed only in refinements, discussed above. We categorized each cause as either a *modeling gap* (78%), reflecting limitations in our workflow's protocol representation, or an *information gap* (22%), reflecting limitations in the available documentation. Figure 5 shows the distribution of root causes. The most common cause is *parameter-conditional transitions* (7 classes), where the agent does not express transitions that depend on method arguments, such as `ChoiceCallback` where the constructor's boolean parameter `multipleSelectionsAllowed` determines the initial state (multiple vs. single selection). The second most common root cause (6 classes) is *getter under-restriction*, where the agent fails to recognize that certain methods need to be restricted (e.g., in `DatagramSocket` the agent does not annotate `getRemoteSocketAddress`, allowing it to be called in any state even if no connection is established). Additionally, the agents never used multiple orthogonal state sets, preferring flat state machines instead. For example, in `ImageWriteParam`, the agent modeled 9 states in a single set rather than using the experts' approach of two orthogonal sets with three states each. All of these cases are *modeling gaps*, as they reflect limitations of the agentic workflow modeling of the protocol. The information gap cases derive from limitations in the documentation, such as *source code dependencies* (e.g., for `DragSourceContext`, experts looked at the documentation to see that there can only be one listener and modeled according to that, while

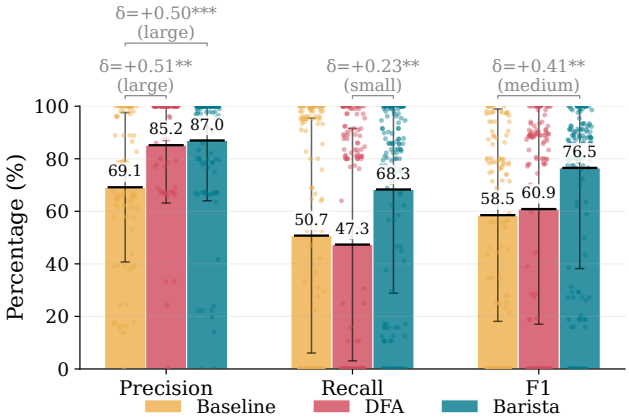

Figure 6: Ablation study comparing pipeline variants, cross-evaluated with expert tests. Bars show aggregated values; error bars indicate per-run standard deviation; dots show individual run results. Brackets annotate significant improvements (Wilcoxon signed-rank, p<0.05) with Cliff's δ effect size; only non-negligible effects are shown. * p<0.05, ** p<0.01, *** p<0.001.

the agentic workflow did not have this information), or *cross-class contracts* (e.g., `RequiredModelMBean` is intended to be used along with another Bean, and experts looked at the interaction between the two from both documentation perspectives).

Despite all these challenges, when looking at the practical usability of the generated specifications, 75% of them are usable as a starting point for expert refinement (29% directly usable, 46% with minor edits). We rated specifications as directly *usable* when states and transitions aligned with expert annotations (mean precision 99.8%, recall 96.6%), as usable *with edits* when the core protocol structure was correct but contained local errors such as missing getter restrictions or argument conditions, and as *not usable* when fundamental gaps such as missing state dimensions required redesigning the concepts in the state machine rather than applying incremental fixes.

Answer to RQ1.2

Generated specifications achieve exact state alignment in 57% of classes. When divergences occur, the agent under-restricts rather than over-restricts, explaining the high precision but moderate recall from RQ1.1. These divergences stem primarily from modeling gaps (78% of root causes) and are more prevalent in complex protocols, yet 75% of specifications remain usable as starting points for refinement.

5.5 Impact of Design Choices (RQ2)

To assess the contribution of key design choices, we implemented two ablation variants and compared them against the full Barista pipeline using the same cross-evaluation protocol with expert tests. The *Baseline* variant uses few-shot prompting without any pipeline components, receiving class documentation and generating specifications directly. The *DFA* variant adds DFA-guided generation with iterative refinement but omits test synthesis.

Figure 6 shows the aggregated metrics across all classes and runs for each variant. We performed Wilcoxon signed-rank tests for each pairwise comparison (Baseline-DFA, DFA-Barista, Baseline-Barista) and computed effect sizes using Cliff's δ. DFA-guided generation significantly improves precision over the baseline with a large effect size, demonstrating that the structured intermediate representation reduces false positives by constraining the agent's protocol modeling. The full Barista pipeline significantly improves recall over the DFA variant with a small effect size, indicating that test-driven validation helps capture more violations by providing concrete usage patterns that guide specification refinement. For F1 score (balancing precision and recall), the full pipeline significantly outperforms the baseline with a medium effect size, confirming that the combination of all components achieves the best overall performance.

To assess cost-effectiveness, we compared Barista's resource usage against human annotators and the pipeline ablations. Barista produces a specification in 7.3 minutes at $2.20 per class, compared to 31 minutes for human annotators, representing a 4.2× speedup. In practical terms, Barista annotates approximately 8 classes per hour at $18, while a human annotator completes roughly 2 classes in the same time. This cost structure makes formal specification synthesis accessible to projects that lack dedicated verification expertise, enabling teams to annotate entire codebases rather than selecting only a few critical classes. Each pipeline stage contributes incrementally to this performance, as the DFA stage adds $0.60 per class and yields a 16 percentage point gain in precision, while the test-verification loop adds $1.38 per class and delivers a 21 percentage point gain in recall. Across all 35 classes and 5 runs, the total cost was $382.

We can further contextualize these cost differences by examining the turns each agent takes and the number of tool iterations it requires. Figure 7 shows that the test completion agent takes the most turns, likely due to the large number of files involved in this stage. The verifier agent follows, as it iteratively refines specifications based on verification feedback, and then the orchestrator that delegates between all agents. The protocol analyzer and specification writer agents take the fewest turns, as they only need to produce a single output file each.

Answer to RQ2

Each pipeline component contributes distinctly to specification quality. DFA-guided generation significantly improves precision over the baseline with a large effect size, while test-driven validation significantly improves recall with a small effect size. The full pipeline achieves these gains at $2.20 per class with a 4.2× speedup over human annotation, making formal specification synthesis practical at scale.

6 Discussion and Future Work

Our results show that Barista produces specifications that are precise enough to be trustworthy and usable enough to serve as starting

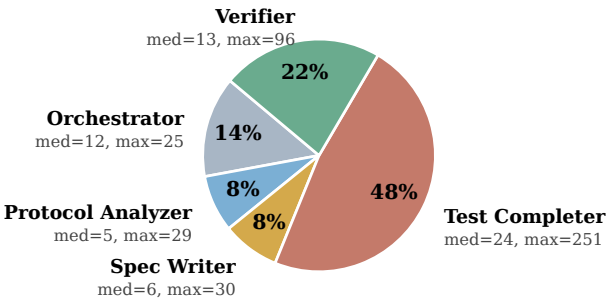

Figure 7: Distribution of agent turns across Barista's subagents including the orchestrator (mean share of total turns per run). Median and maximum turn counts shown per agent.

points for expert refinement. In this section, we discuss the practical implications of these findings and identify concrete directions for improvement grounded in our empirical observations.

Intermediate Representation for Precision, Tests for Recall. Our ablation study isolates two factors behind Barista's performance: the DFA intermediate representation and the test-driven verification loop. The intermediate representation constrains how the agent models the protocol, providing structured guidance that significantly increases precision and reduces false positives. The verification loop then supplies concrete usage patterns that broaden the cases the specification must cover, significantly raising recall and reducing false negatives. Together, these components let Barista strike a better balance between precision and recall than either achieves alone. We orchestrate them with a multi-agent framework, but the gains come from the pipeline and its intermediate representation, more than from the orchestration itself.

Precision as a Practical Priority. Research on static analysis adoption consistently shows that false positives are a primary driver of tool abandonment [16, 41]. Barista's 87% precision means that the vast majority of flagged violations are real, which is critical for maintaining developer trust in the tool's recommendations. The moderate recall (68.3%) positions our approach as a bootstrapping aid rather than an expert replacement, as developers can take the generated specification as a starting point that already captures the majority of protocol violations, then refine it to catch the remaining cases. Our qualitative analysis supports this framing, as 75% of generated specifications are usable directly or with minor edits, and the 4.2× speedup over manual annotation makes it practical to generate specifications for entire codebases rather than only a handful of critical classes. However, given the limitations of the current approach, we recommend two mitigation strategies for practitioners. First, treat generated specifications as a starting point and prioritize manual review of complex protocols, using our complexity rubric to help identify which classes warrant additional scrutiny. Second, complement Barista-generated specifications with runtime monitoring during testing to catch violations the static specifications may miss.

The specific root causes we identified also provide a concrete roadmap for improving Barista's specification synthesis.

The Expressiveness Bottleneck. The most common root cause of specification divergences comes from the parameter-conditional transitions, which our DFA intermediate representation cannot express. Although the DFA formalism significantly improves precision by constraining the agent's modeling, it creates a bottleneck for protocols where transitions depend on method argument values. This is a modeling gap of our approach, not an intelligence gap of the LLM. Future work can address this limitation by exploring richer intermediate representations such as symbolic automata [49] or extended finite-state machines that can directly express parameter-conditional transitions, which would allow the agent to capture a larger portion of the protocol logic without sacrificing the structured guidance that improves precision. Additionally, modal transition systems [14, 19] are a complementary target as their may/must transitions could represent the partial knowledge that leads Barista to under-restrict more often than it over-restricts. Addressing this single limitation would improve specification quality for 7 of the 22 divergent classes, making it a high-impact target for subsequent iterations of the pipeline.

Limits of Documentation as the Unique Source. Our results validate the initial insight that class documentation contains sufficient protocol knowledge to generate useful typestate specifications. Barista successfully detected protocols in 28 of 29 classes and produced usable specifications for 75% of them, demonstrating that Javadoc captures the majority of protocol logic developers need. These results, however, depend on documentation being available, which is an explicit precondition of our approach. In practice, this requirement aligns with where protocol checking has the highest impact, since the most mature and widely used libraries are typically the best documented. By annotating a single such library, like the JDK, we enable compile-time protocol checking for every client that consumes it.

However, even with mature documentation, our qualitative results reveal cases where documentation alone is insufficient. Experts needed to consult source code and examine other classes that interact with the target class to resolve ambiguities and capture cross-class contracts. These dependencies are common in complex protocols but rarely documented in a single class's Javadoc, contributing to 4 of the 22 divergent specifications in our dataset. This suggests that while good documentation provides a valuable starting point, augmenting it with lightweight source code analysis could help surface implicit state transitions and inter-class dependencies that Javadoc omits. Nevertheless, experts also had divergent specifications, suggesting that some gaps reflect inherent ambiguity in the documentation rather than agent limitations. Addressing these cases may require clearer documentation practices or more sophisticated interpretive heuristics rather than simply providing more information to the agent.

Error Messages and the Feedback Loop. The test-driven verification loop significantly improves recall by providing concrete usage patterns that guide specification refinement. However, this loop can also degrade individual specifications when the agent cannot interpret the verification feedback, as observed in the `PipedWriter` case where uninformative error messages led the agent to remove correct refinements in order to achieve successful verification. Verification tools are known to produce error messages that challenge

even human developers [22], and our findings suggest this problem compounds for agentic workflows. This opens an interesting research direction on designing agent-oriented verification feedback. Making errors actionable not just for humans but also for LLM agents could improve the reliability of verification-driven synthesis pipelines beyond our specific application to typestate protocols.

6.1 Threats to Validity

Internal Validity. Expert annotations may introduce bias, as annotators might interpret ambiguous documentation differently or impose their own mental models onto the specifications. We mitigated this threat through dual independent annotation followed by reconciliation meetings where annotators discussed disagreements and converged on a single specification. Additionally, we documented decision rationales for all ambiguous cases, making our interpretive choices transparent and reproducible.

External Validity. Our dataset comprises 35 classes drawn from a single empirical study on object protocols, all from the JDK standard library. While this selection ensures well-documented APIs and established usage patterns, it may not generalize to application-level code with sparser documentation or to languages with different object models. Moreover, JDK classes are heavily represented in LLM training corpora, so the model may benefit from prior exposure to these protocols, potentially inflating performance relative to less well-known APIs [37]. This concern is partially mitigated by the fact that our task requires generating LiquidJava annotations, a niche format unlikely to appear in training data. Additionally, the protocol patterns we target (initialization, liveness checks, deactivation) are architectural patterns that recur across codebases. We expect similar precision on well-documented third-party libraries, but classes with sparse or missing Javadoc would likely see degraded recall. Future work should evaluate Barista on popular open-source libraries such as Apache Commons or Google Guava to validate this hypothesis.

Construct Validity. Our strategy of averaging the results of all runs of the workflow (micro-averaging) gives more weight to classes with larger test suites, which may skew aggregate metrics toward classes where test generation succeeded most comprehensively. Additionally, the qualitative analysis of agent logs and comparison between agentic and expert annotations was conducted by the authors rather than independent reviewers, potentially introducing interpretive bias in categorizing failure modes. Finally, the same object protocol can be modeled in several ways, so two expert specifications may capture identical behavior yet differ in representation. Structural metrics such as state count consequently understate annotator agreement, which is why we anchor our evaluation on behavioral agreement over the test suite.

Non-Determinism. LLM-based generation introduces inherent variability across runs. We executed 5 runs per class to capture this variance and report both aggregate metrics and per-class consistency, but this sample may not exhaustively characterize the distribution of possible outputs.

7 Conclusion

We presented Barista, an agentic workflow that transforms Java class documentation into verifiable LiquidJava specifications encoding object state protocols and method-level constraints. On a benchmark of 35 well-documented Java classes with expert-written ground truth, Barista achieves 87.0% precision and 68.3% recall, with 75% of generated specifications usable directly or with minor edits, making it an initial step towards practical formal specification for entire codebases. The key pipeline contributors to this performance are the DFA intermediate representation that drives precision, and the test-driven verification loop that drives recall, as shown in our ablation study. Additionally, the qualitative analysis reveals that remaining gaps stem from modeling limitations rather than LLM reasoning failures, providing a concrete roadmap for future improvements.

The specification burden has been a limitation for adoption of formal verification by teams without dedicated expertise. Our results demonstrate that agentic workflows can automate the most labor-intensive steps of producing specifications that developers can use as a bootstrap and refine. As LLM capabilities continue to improve and verification tools become more accessible, approaches like Barista can help bring compile-time safety guarantees to development teams that currently lack the resources to write formal specifications manually.

8 Data Availability Statement

We provide a replication package with all the data and code to reproduce our results [7]. The full implementation of Barista, along with the dataset of expert annotations, generated specifications and tests, is available in the package. We also provide detailed instructions for the manual annotation and qualitative analysis processes. The scripts for analysis of the results, including the generation of all figures and tables, are also included to facilitate reproducibility and further exploration by other researchers.

Acknowledgments

This work is funded by national funds through FCT – Fundação para a Ciência e a Tecnologia, I.P., under the project CMU-Portugal, ref. SFRH/BD/151469/2021 and PRT/BD/154254/2021, and under the LASIGE Research Unit, ref. UID/00408/2025, DOI https://doi.org/10.54499/UID/00408/2025, which includes the LASIGE Seed project ref. LASIGE-SP-2025.07.

This work was also funded by the European Union - NextGenerationEU through the PRR - Recovery and Resilience Plan under the LASIGE Research Unit, ref. UID/PRR/00408/2025, DOI https://doi.org/10.54499/UID/PRR/00408/2025 and ref. UID/PRR2/00408/2025.

We used AI-assisted tools during the preparation of this work. Specifically, we used Claude Opus 4.6 to assist in generating data analysis scripts and figures, and to improve the wording and fluency of the paper.

References

[1] Jonathan Aldrich, Robert Bocchino, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. 2011. Plaid: a permission-based programming language. In *International Conference on Object Oriented Programming Systems Languages and Applications*. doi:10.1145/2048147.2048197

[2] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-oriented programming. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. doi:10.1145/1639950.1640073

[3] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1170–1188. doi:10.1109/TSE.2018.2827384

[4] Anthropic. 2025. Agent SDK overview. https://platform.claude.com/docs/en/agent-sdk/overview Accessed: March 26, 2026.

[5] Anthropic. 2025. Claude Code. https://www.claude.com/product/claude-code Accessed: March 26, 2026.

[6] Anthropic. 2025. Skills. https://code.claude.com/docs/en/skills Accessed: March 26, 2026.

[7] Barista. 2026. Artifact for Barista: Synthesizing Typestate Specifications with LLM Agents. https://figshare.com/s/3126cf30b149a9114e05. Accessed: March 26, 2026.

[8] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. 2011. An Empirical Study of Object Protocols in the Wild. In *ECOOP 2011 - Object-Oriented Programming*. doi:10.1007/978-3-642-22655-7_2

[9] Kevin Bierhoff and Jonathan Aldrich. 2008. PLURAL: checking protocol compliance under aliasing. In *International Conference on Software Engineering*. doi:10.1145/1370175.1370213

[10] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. 2009. Practical API Protocol Checking with Access Permissions. In *ECOOP 2009 - Object-Oriented Programming*. doi:10.1007/978-3-642-03013-0_10

[11] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu K. Lahiri, and Nikhil Swamy. 2025. Towards Neural Synthesis for SMT-Assisted Proof-Oriented Programming. In *International Conference on Software Engineering*. doi:10.1109/ICSE55347.2025.00002

[12] Daniela S. Cruzes and Tore Dybå. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *Empirical Software Engineering and Measurement*. doi:10.1109/ESEM.2011.36

[13] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. doi:10.1007/978-3-540-78800-3_24

[14] Nicolás D'Ippolito, Dario Fischbein, Marsha Chechik, and Sebastián Uchitel. 2008. MTSA: The Modal Transition System Analyser. In *International Conference on Automated Software Engineering*. doi:10.1109/ASE.2008.78

[15] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* (2019). doi:10.1145/3338112

[16] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. 2022. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering* 48, 3 (2022), 835–847. doi:10.1109/TSE.2020.3004525

[17] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE (2024), 1889–1912. doi:10.1145/3660791

[18] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming* (2007). doi:10.1016/j.scico.2007.01.015

[19] Dario Fischbein, Sebastián Uchitel, and Víctor A. Braberman. 2006. A foundation for behavioural conformance in software product line architectures. In *Workshop on Role of Software Architecture for Testing and Analysis, held in conjunction with the International Symposium on Software Testing and Analysis*. doi:10.1145/1147249.1147254

[20] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *Formal Methods for Increasing Software Productivity*. doi:10.1007/3-540-45251-6_29

[21] Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *International Conference on Software Engineering*. doi:10.1109/ICSE48619.2023.00132

[22] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. *Proc. ACM Program. Lang.* PLDI (2025). doi:10.1145/3729327

[23] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. 2024. Property-Based Testing in Practice. In *International Conference on Software Engineering (ICSE)*. Article 187. doi:10.1145/3597503.3639581

[24] Falk Howar and Bernhard Steffen. 2018. Active Automata Learning in Practice - An Annotated Bibliography of the Years 2011 to 2016. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar*. doi:10.1007/978-3-319-96562-8_5

[25] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations ICLR*. doi:10.48550/arXiv.2310.06770

[26] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing. In *International Symposium on Software Testing and Analysis (ISSTA)*. doi:10.1145/3293882.3330552

[27] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Sci. Comput. Program.* (2018). doi:10.1016/j.scico.2017.10.006

[28] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. In *International Conference on Automated Software Engineering (ASE)*. doi:10.1145/2970276.2970356

[29] Jinghua Liu, Yi Yang, Kai Chen, and Miaoqian Lin. 2025. Generating API Parameter Security Rules with LLM for API Misuse Detection. In *Network and Distributed System Security Symposium, NDSS*. doi:10.14722/ndss.2025.230465

[30] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *International Conference on Software Engineering*. doi:10.1109/ICSE55347.2025.00129

[31] Kane Meissel and Esther S. Yao. 2024. Using Cliff's Delta as a Non-Parametric Effect Size Measure: An Accessible Web App and R Tutorial. *Practical Assessment, Research, and Evaluation* (2024). doi:10.7275/pare.1977

[32] João Mota, Marco Giunti, and António Ravara. 2021. Java Typestate Checker. In *Coordination Models and Languages held as part of the International Federated Conference on Distributed Computing Techniques, DisCoTec*. doi:10.1007/978-3-030-78142-2_8

[33] Christian Muise, Sheila A. McIlraith, and J. Christopher Beck. 2012. Improved non-deterministic planning by exploiting state relevance. In *International Conference on Automated Planning and Scheduling*. doi:10.5555/3038546.3038567

[34] Francisco Oliveira, Alexandra Mendes, and Carolina Carreira. 2025. What Challenges Do Developers Face When Using Verification-Aware Programming Languages?. In *International Symposium on Software Reliability Engineering*. doi:10.1109/ISSRE66568.2025.00031

[35] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *International Conference on Software Engineering (ICSE)*. doi:10.1109/ICSE.2012.6227127

[36] Daniel Ramos, Catarina Gamboa, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2026. SPELL: Synthesis of Programmatic Edits using LLMs. *CoRR* abs/2602.01107 (2026). doi:10.48550/ARXIV.2602.01107

[37] Daniel Ramos, Cláudia Mamede, Kush Jain, Paulo Canelas, Catarina Gamboa, and Claire Le Goues. 2025. Are Large Language Models Memorizing Bug Benchmarks?. In *International Workshop on Large Language Models for Code, LLM4Code@ICSE*. 1–8. doi:10.1109/LLM4CODE66737.2025.00005

[38] Cedric Richter and Heike Wehrheim. 2025. Beyond Postconditions: Can Large Language Models Infer Formal Contracts for Automatic Software Verification? *arXiv preprint* (2025). doi:10.48550/arXiv.2510.12702

[39] Patrick Maxim Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Programming Language Design and Implementation (PLDI)*. ACM, 159–169. doi:10.1145/1375581.1375602

[40] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* 14, 2 (2009), 131–164. doi:10.1007/S10664-008-9102-8

[41] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66. doi:10.1145/3188720

[42] Johnny Saldaña. 2009. *The Coding Manual for Qualitative Researchers*. SAGE Publications.

[43] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Eng.* 25, 4 (1999), 557–572. doi:10.1109/32.799955

[44] C Spearman. 2010. The proof and measurement of association between two things. *International Journal of Epidemiology* 39, 5 (10 2010), 1137–1150. arXiv:https://academic.oup.com/ije/article-pdf/39/5/1137/18481215/dyq191.pdf doi:10.1093/ije/dyq191

[45] Robert E. Strom and Shaula Yemini. 1986. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Trans. Software Eng.* 12, 1 (1986), 157–171. doi:10.1109/TSE.1986.6312929

[46] Chuyue Sun, Viraj Agashe, Saikat Chakraborty, Jubi Taneja, Clark W. Barrett, David L. Dill, Xiaokang Qiu, and Shuvendu K. Lahiri. 2025. ClassInvGen: Class Invariant Synthesis Using Large Language Models. In *AI Verification - International Symposium, SAIV*. doi:10.1007/978-3-031-99991-8_4

[47] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2014. Structuring Documentation to Support State Search: A Laboratory Experiment about Protocol Programming. In *ECOOP 2014 - Object-Oriented Programming*. doi:10.1007/978-3-662-44202-9_7

[48] Joshua Sunshine, James D. Herbsleb, and Jonathan Aldrich. 2015. Searching the State Space: A Qualitative Study of API Protocol Usability. In *International Conference on Program Comprehension*. doi:10.1109/ICPC.2015.17

[49] Hellis Tamm and Margus Veanes. 2018. Theoretical Aspects of Symbolic Automata. In *SOFSEM 2018: Theory and Practice of Computer Science - International Conference on Current Trends in Theory and Practice of Computer Science*. doi:10.1007/978-3-319-73117-9_30

[50] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2023. Can Large Language Models Write Good Property-Based Tests? *CoRR* abs/2307.04346 (2023). doi:10.48550/ARXIV.2307.04346

[51] Haiming Wang, Huajian Xin, Chuanyang Zheng, Zhengying Liu, Qingxing Cao, Yinya Huang, Jing Xiong, Han Shi, Enze Xie, Jian Yin, Zhenguo Li, and Xiaodan Liang. 2024. LEGO-Prover: Neural Theorem Proving with Growing Libraries. In *International Conference on Learning Representations, ICLR*. doi:10.48550/arXiv.2310.00656

[52] Haiyang Wei, Ligeng Chen, Zhengjie Du, Yuhan Wu, Haohui Huang, Yue Liu, Guang Cheng, Fengyuan Xu, Linzhang Wang, and Bing Mao. 2025. Unleashing the Power of LLM to Infer State Machine From the Protocol Implementation. In *International Symposium on Quality of Service, IWQoS*. doi:10.1109/IWQOS65803.2025.11143461

[53] Westley Weimer and George C. Necula. 2005. Mining temporal specifications for error detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. doi:10.1007/978-3-540-31980-1_30

[54] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification. In *Computer Aided Verification CAV*. doi:10.1007/978-3-031-65630-9_16

[55] John Whaley, Michael C. Martin, and Monica S. Lam. 2002. Automatic extraction of object-oriented component interfaces. *SIGSOFT Softw. Eng. Notes* 27, 4 (July 2002), 218–228. doi:10.1145/566171.566212

[56] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. doi:10.2307/3001968

[57] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu K. Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. 2025. AutoVerus: Automated Proof Generation for Rust Code. *Proc. ACM Program. Lang.* (2025). doi:10.1145/3763174

[58] Miao Zhang, Runhan Feng, Hongbo Tang, Yu Zhao, Jie Yang, Hang Qiu, and Qi Liu. 2025. Automated Extraction of Protocol State Machines from 3GPP Specifications with Domain-Informed Prompts and LLM Ensembles. *CoRR* abs/2510.14348 (2025). doi:10.48550/ARXIV.2510.14348

[59] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. 2020. An empirical study on API parameter rules. In *International Conference on Software Engineering (ICSE)*. doi:10.1145/3377811.3380922

Received 2026-03-26; accepted 2026-06-18