USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Sharpen the Spec, Cut the Code: A Case for Generative File System with SysSpec

Qingyuan Liu, Mo Zou, Hengbin Zhang, Dong Du, and Yubin Xia, and Haibo Chen, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/fast26/presentation/liu-qingyuan

This paper is included in the Proceedings of the 24th USENIX Conference on File and Storage Technologies.

February 24–26, 2026 • Santa Clara, CA, USA

ISBN 978-1-939133-53-3

Open access to the Proceedings of the 24th USENIX Conference on File and Storage Technologies is sponsored by

Sharpen the Spec, Cut the Code: A Case for Generative File System with SysSpec

Qingyuan Liu, Mo Zou, Hengbin Zhang, Dong Du, Yubin Xia, Haibo Chen
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University

Abstract

File systems are critical OS components that require constant evolution to support new hardware and emerging application needs. However, the traditional paradigm of developing features, fixing bugs, and maintaining the system incurs significant overhead, especially as systems grow in complexity. This paper proposes a new paradigm, *generative file systems*, which leverages Large Language Models (LLMs) to generate and evolve a file system from prompts, effectively addressing the need for robust evolution. Despite the widespread success of LLMs in code generation, attempts to create a functional file system have thus far been unsuccessful, mainly due to the ambiguity of natural language prompts.

This paper introduces SysSpec, a framework for developing generative file systems. Its key insight is to *replace ambiguous natural language with principles adapted from formal methods*. Instead of imprecise prompts, SysSpec employs a multi-part *specification* that accurately describes a file system's functionality, modularity, and concurrency. The specification acts as an unambiguous blueprint, guiding LLMs to generate expected code flexibly. To manage evolution, we develop a *DAG-structured patch* that operates on the specification itself, enabling new features to be added without violating existing invariants. Moreover, the SysSpec toolchain features a set of LLM-based agents with mechanisms to mitigate hallucination during construction and evolution. We demonstrate our approach by generating SpecFS, a concurrent file system. SpecFS demonstrates equivalent level of correctness to that of a manually-coded baseline across hundreds of regression tests. We further confirm its evolvability by seamlessly integrating 10 real-world features from Ext4. Our work shows that a specification-guided approach makes generating and evolving complex systems not only feasible but also highly effective.

1 Introduction

File systems are a cornerstone of modern operating systems, providing the critical abstractions for managing persistent data. Their design is dictated by a symbiotic relationship with two forces: the characteristics of underlying storage hardware and the demands of ever-changing applications. This relationship necessitates continuous evolution, driving the creation of specialized file systems like F2FS for flash memory [29] and EROFS for read-only mobile scenarios [22]. Consequently, file system developers are in a perpetual cycle of adding features, optimizing performance, and resolving bugs to keep pace with innovation.

However, this evolution comes at a steep price. To quantify this challenge, we conduct a longitudinal study of the Ext4 file system's development, analyzing all 3,157 commits from its inception in Linux 2.6.19 to the recent 6.15 release. Our analysis reveals that 82.4% of all commits are dedicated to bug fixes and maintenance, which are a direct consequence of introducing new functionality (which accounts for only 5.1% of total commits). E.g., the recently merged "fast commits" feature [41] required only 9 commits for its initial implementation, while it triggered about 80 subsequent commits to address newly introduced bugs and maintain the code (§2.2). This demonstrates a common development cycle where the effort to stabilize new features far outweighs the initial implementation effort, placing a non-trivial burden on developers.

This motivates us to explore a new paradigm for file system design and development, *generative file systems*, which leverages the capabilities of Large Language Models (LLMs) [1, 11, 33, 34, 48] to generate a complete file system and evolve it effectively.

Nevertheless, although recent advancements in LLMs have demonstrated their profound capabilities in automated code generation [2, 3, 20, 24, 25, 49, 53, 54], generating a complete and useful file system from natural-language prompts is profoundly challenging, if not impossible. The intricate semantics of file systems, from ensuring concurrency correctness to carefully managing complex file structures, are difficult to express unambiguously in prompts. Consequently, to our knowledge, no prior LLM-based approach has successfully generated a complete, functional, and feature-rich file system.

Compared with descriptions with natural language, *specifications* [38, 45] usually provide a precise and machine-understandable language to *explicitly encode* these invariants (e.g., no orphan inodes after a crash), creating an opportunity to guide LLMs toward generating expected code. However, bridging the gap between specifications and LLM-based code generation is non-trivial. We identify three technical challenges that our work, SysSpec, is designed to overcome:

Challenge I: Specification semantic gap. A specification must be expressive enough to capture the multifaceted semantics of a file system, which natural language prompts fail to do. This includes not only functional correctness but

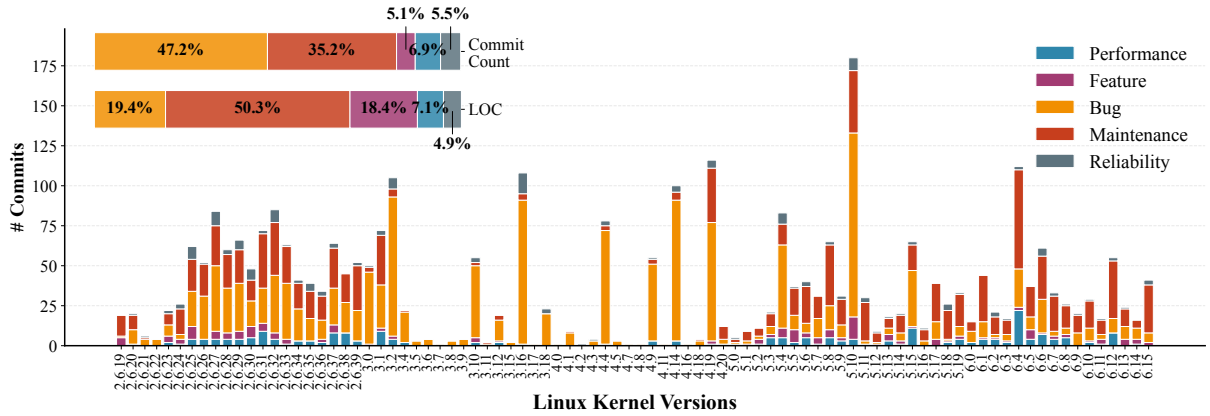


Figure 1: **File system evolution with different types of patches**.

also non-functional properties critical for performance, such as on-disk layout choices (e.g., bitmap vs. linear scan for block allocation). Furthermore, precisely specifying complex concurrency control is notoriously difficult. Attempting to describe both functional logic and fine-grained locking in a single, monolithic prompt can overwhelm an LLM, causing it to overlook subtle but critical details and produce code with concurrency bugs.

Challenge II: Complex component composition. The finite context window of LLMs precludes generating an entire file system monolithically. This necessitates a modular approach, which introduces significant composition challenges. First, generating one module at a time, an LLM lacks the global context to ensure interface compatibility with other, yet-to-be-generated or pre-existing components, leading to integration errors. Second, each individual change (with LLMs) introduced during the evolution process can potentially trigger cascading effects on other file system modules, particularly those with existing dependencies. For example, a seemingly local feature addition, like introducing extent [4], can trigger non-local changes by altering core data structures like the inode, affecting any module that interacts with it and making manual dependency management intractable.

Challenge III: Unreliable LLM capability. LLM-based code generation is inherently non-deterministic due to the hallucination [26, 50] — even identical specifications can yield different and potentially incorrect code outputs across generation attempts. A naive "generate-and-pray" approach is unacceptable for system software. Therefore, a robust framework cannot blindly trust the LLM especially for file systems; it must incorporate a rigorous validation mechanism to ensure that generated code strictly adheres to the guiding specification, guaranteeing correctness despite the unreliability.

To address these challenges, our key insight is to *replace ambiguous natural language with specifications following the principles adapted from successful practices of formal methods* [15, 42, 55]. Specifically, we introduce SysSpec, a framework for correct-by-construction file system generation built on three core techniques.

First, we design a **formal method-inspired specification** to precisely define a file system's behavior, moving beyond the ambiguity of natural language. It holistically captures the FS design, spanning **Functionality**, defined through Hoare-logic based pre/post-conditions [15, 55] and invariants; **Modularity**, which enforces a clean decomposition into modules with rely-guarantee [31, 55] interfaces for correct composition; and **Concurrency**, which makes locking protocols and ordering explicit to mitigate subtle bugs.

Second, SysSpec provides an *LLM-based toolchain* that translates the high-level specification into an executable C implementation. The toolchain contains three LLM-based agents: the SpecCompiler, which systematically synthesizes C code from the specification using techniques like *two-phase generation* (logic first, then concurrency); the SpecValidator, which rigorously validates the generated code against the specification and drives a *retry-with-feedback* loop to autonomously correct LLM errors; and then SpecAssistant which eases the development of specification.

Third, we introduce a mechanism for **principled evolution via spec patches**. Instead of manually modifying C implementation, developers add features by authoring a special patch, called *spec patch*, to the high-level specification. The SysSpec toolchain then automatically propagates this change, regenerating the implementation to ensure the new feature is correctly and consistently integrated.

Overall, the new paradigm with SysSpec shifts the developer's burden from low-level implementation to high-level design. While this demands greater upfront design effort, akin to the safety discipline of Rust [13, 27], the return is a file system that is far easier to maintain and evolve.

We utilize SysSpec to implement SpecFS, a complete FUSE-based (generative) file system specified entirely in specification. Our toolchain automatically generates a functional C-language implementation without any manual intervention. We validate its correctness and expressiveness by successfully generating implementations of a previously verified file system, AtomFS [55]. Besides, we showcase its capability for complex feature integration by seamlessly evolving

SPECFS with spec patches to support 10 novel features from Ext4, including delayed allocation and file encryption. We also highlight its benefits on performance: by applying Ext4-style delayed allocation method with a concise spec patch, SYSSPEC automatically regenerated the relevant modules, resulting in a 99.9% data write reduction for xv6 compilation.

To our knowledge, SYSSPEC is the first framework that enables both the generation and principled evolution of end-to-end file systems like SPECFS, shifting the developer's focus from writing brittle low-level code to crafting robust, high-level designs. SYSSPEC and SPECFS is available at: https://llmnativeos.github.io/specfs/.

2 Characterizing the Evolution of File Systems

File systems usually require a continuous evolution to adapt to new hardware features and emerging use cases. This constant adaptation includes adding new features, resolving bugs, and optimizing for critical scenarios. Understanding the intricate process of file system evolution offers valuable insights for future design. Prior work by L. Lu et al. [35] in 2013 provided the first analysis of the Linux file system's evolution. We continue the analysis with an extended study of evolution over a 20-year period (from 2005 to 2025), and identify new implications that highlight the need to re-evaluate current file system paradigms, particularly in the context of the new opportunities provided by large language models.

2.1 The Anatomy of File System Change

Methodology. We choose Ext4 as our target because it is a mature and (still) widely-deployed file system with 20 years of evolution in real-world environments. Ext4 was introduced in Linux 2.6.19 and has been continuously developed, incorporating thousands of patches related to performance, new features, and maintenance. Our analysis is based on all 3,157 Ext4-related commits (i.e., individual patches rather than patch sets) merged into the mainline Linux from version 2.6.19 to 6.15. To understand the evolution, we categorize each commit using a classification scheme for FS patches adapted from prior work [36]: (1) Bug: fixing an existing bug, (2) Performance: improving efficiency through new designs or optimizations, (3) Reliability: enhancing the file system's robustness, (4) Feature: implementing new functionalities, and (5) Maintenance: refactoring code or improving documentation without changing semantic behavior.

Implication-1: File systems consistently evolve. Our analysis of Ext4 patches across Linux versions, as shown in Fig. 1, reveals a clear evolutionary trend. Initially, during the early stages (Linux 2.6.19–3.4), the high number of changes reflects extensive work on new features, bug fixes, and maintenance. The number of changes then decreases significantly from Linux 3.4 to 4.18 as the codebase matures. However, a surprising trend emerges: changes increase steadily after Linux 4.19, peaking at Linux 5.10 — more than a decade

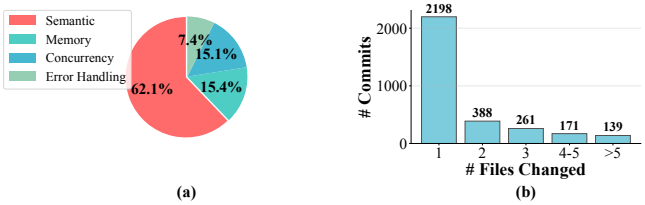


Figure 2: **(a) Distribution of bug type. (b) Distribution of changed files per commit.**

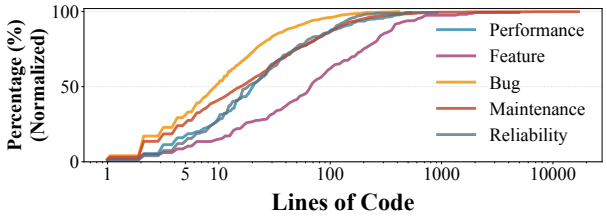


Figure 3: **Patch LOC size CDF**.

after Ext4's introduction. We also observe occasional peaks in the number of changes even during the stable period (Linux 2.6.19–3.4), such as over 50 changes in Linux 3.10 and over 100 in Linux 3.16. This result suggests that a long-lived file system like Ext4 is in *a state of constant evolution*.

Implication-2: Bug fixes and maintenance dominate a file system's lifetime. While users are often drawn to a file system by its key features, e.g., Ext4's use of extents, our analysis reveals that non-feature patches dominate the file system's lifetime. Specifically, 82.4% of all commits focus on bug fixes and maintenance. Fig. 2-a further shows the distribution of bug type, indicating that most bugs are semantic bugs. This finding presents an insight: although new features initially attract users, the long-term success and widespread adoption of a FS depend heavily on its continuous maintenance. However, this high volume of maintenance work represents a significant and ongoing burden, and current paradigms lack efficient solutions to handle it.

Implication-3: Feature-related changes are non-trivial. While feature-related commits constitute only 5.1% of the total changes, their impact is significant. Our analysis of the code base, as shown in Fig. 1, reveals that these commits account for 18.4% of the total lines of code (LOC) changed. Furthermore, we observe that the introduction of a new feature often acts as a catalyst for subsequent bug fixes and maintenance work. A new feature introduces a new code base, which in turn necessitates follow-up commits to fix newly introduced bugs and maintain the code's integrity. Consequently, despite their low commit count, feature-related changes are central to the overall evolution of a file system.

Implication-4: Evolution is taken in small steps. We measured the LOC for each patch and present the cumulative distribution function (CDF) in Fig. 3. Our analysis shows that evolution proceeds through small, frequent changes. Specifically, about 80% of all bug fixes involve fewer than 20 LOC. While feature-related patches are generally larger, about 60% of them still require fewer than 100 LOC. Moreover, the vast

```
//  1. Internal logical error.
-- if (unlikely(error))
++ if (unlikely(error)) {
++    ext4_fc_stop_update(inode);
      return error;
++ }
//  2. Cross-module collision.
-- #define EXT4_MOUNT2_DAX_INODE 0x000010
   #define EXT4_MOUNT2_JOURNAL_FAST_COMMIT
   ↪  0x000010
++ #define EXT4_MOUNT2_DAX_INODE 0x000040
```

Figure 4: **Two Example Patches in `fast-commit`.**

majority of commits also modify only a single file, as shown in Fig. 2-b. These findings suggest that file system evolution is composed of these manageable, small-scale changes.

2.2 Case Study: Evolution of Fast Commit

We present a case study to illustrate how different types of patches are intertwined during the evolution. We use *fast commit* [41], a hybrid journaling feature designed to optimize `fsync()`-intensive workloads, as an example. Fast commit reduces I/O overhead and latency by using lightweight, logical commits, while periodically issuing full commits to maintain consistency. It is one of the largest features introduced in Linux 5.10, comprising over 4,000 SLOC.

Phase-1: Feature development. Our analysis covers 98 fast-commit-related patches from Linux kernel 5.10 to 6.15. The initial implementation is concentrated in version 5.10, which includes 9 of the 10 feature-related commits. These initial patches collectively introduce `jbd2` APIs for fast-commit support, initialization and recovery paths, and the main commit logic. In total, these commits add over 4,000 lines of code spanning multiple core modules (e.g., inode, file, journal). Despite affecting several modules, the modifications are carefully localized to minimize interference. This modular design preserves the existing journaling mechanism and on-disk format while integrating the new fast-commit logic.

Phase-2: Bug fixes and stabilization. Development efforts shift to stabilization after the initial release. Of the 55 bug-fix commits we have identified, over 65% address semantic errors, e.g., misordered updates and incorrect handling of corner cases. This high proportion of semantic bugs can be attributed to the feature's complexity and its deep integration with other Ext4 components. We classify these bugs as either **internal** (within the fast-commit logic) or **cross-module** (from interactions with other parts of Ext4). Fig. 4 shows two examples. The first illustrates an internal bug where an early return path omitted necessary cleanup operations, leading to lost metadata updates. The second shows a cross-module bug where newly defined flags conflicted with existing journal checksum bits, requiring a redefinition of mount macros to resolve the collision. These examples highlight the inherent difficulty of integrating features that span multiple modules, underscoring the need for careful reasoning about global system invariants.

Phase-3: Code maintenance. Alongside bug fixes, 24 maintenance commits (totaling 1,080 lines) are applied to refactor and document the fast-commit implementation. Examples include: (1) **Refactoring for readability**, where logic for updating statistics is extracted into a dedicated function, `ext4_fc_update_stats()`, to simplify the main `ext4_fc_commit()` pathway. (2) **API clarification**, which involves enhancing flag descriptions in both the source code and documentation to prevent misconfiguration.

The lifecycle of the fast-commit feature exemplifies a core challenge in the evolution of file systems: the initial integration of a feature is usually followed by a long tail of numerous, fine-grained fixes and refactoring efforts that are crucial for stability and long-term maintainability.

Table 1: **Prior code generation methods.** *"0 to N" denotes generating code from scratch, while "N to N+1" refers to generating code based on existing code, representing two categories of current work.*

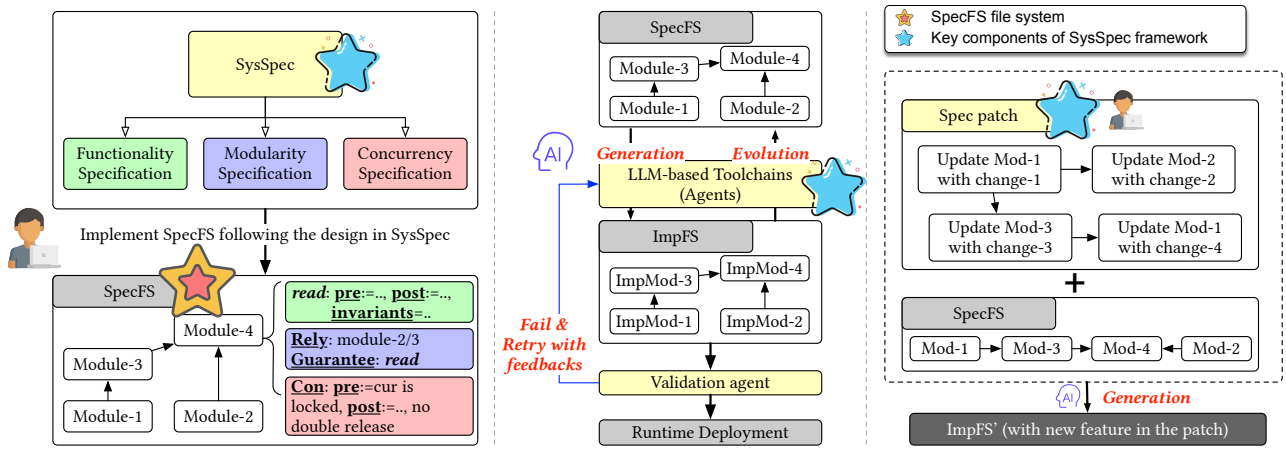
Type	Prior works	Precise	Modular	Concurrent	Specification
0 to N	Copilot [2]	✗	✓	✗	Natural Language
	Clover [46]	✓	✗	✗	Docstring + Annotation
	Qimeng [20]	✓	✗	✗	Programming Language
N to $N+1$	AutoCodeRover [54]	✗	✓	✗	Github Issue
	CodeAgent [53]	✗	✓	✗	Natural Language
	"Intention" [24]	Half	✗	✗	Natural Language
	SpecFS	✓	✓	✓	SysSpec + Toolchain

2.3 A New Paradigm: Generative File System

The previous analysis reveals that FS development involves not only the inherent complexity of feature development but also the long tail of maintenance and bug fixes. This entire process is laborious and error-prone when performed manually in low-level C code. Luckily, the recent advances in Large Language Models (LLMs) offer an opportunity to address this challenge. LLMs' proficiency in code generation, refactoring, and reasoning is well-suited for the high volume of maintenance and bug-fix tasks that consume the majority of developer effort. Furthermore, the scope of code when evolving a file system could typically fit within the context windows of modern LLMs (Implication IV, §2.1), making an automated approach technically feasible.

To this end, we propose the new paradigm of *generative file system*, i.e., leveraging LLMs to generate and evolve a file system. The central thesis of our work is that: *the path forward is not to replace developers with LLMs, but to elevate their role by changing how they express design.*

Limitations of prior works. Although abundant prior works achieve automatic code generation, their inherent limitations prevent us from leveraging them to achieve the paradigm for complex file systems, as generalized in Tab. 1. These works can be broadly categorized into two types. First, some works focus on generating complete code logic from scratch ("from 0 to N"). However, this category of work struggles to cope with the complexity of file systems, whose modules exhibit intricate interactions and dependencies. Consequently, such approaches either produce only simple code (e.g., frontend pages) or are limited to specific single-module tasks [20, 46],



(a) Design Cycle (b) Runtime Workflow (c) Evolution with Spec Patch

Figure 5: **Design overview.** (a) Developers write a file system (SPECFS) using a structured specification language that defines functionality, modularity, and concurrency. (b) An LLM-based toolchain generates a low-level implementation (`ImpFS`) from the specification and uses a validation agent to ensure correctness. (c) The file system evolves by applying a high-level *spec patch* to SPECFS, after which the toolchain regenerates the implementation to include the new features or fixes.

such as generating tensor program implementations for different hardware [20], generating unit tests [14,43], mathematical library functions [12], or performing bit-vector synthesis [19].

Second, other approaches usually focus on evolution, i.e., modifying existing code ("from N to $N+1$"). They may utilize methods like Retrieval-Augmented Generation (RAG) [24,49] or Agents [2,3,24,25,53,54] to enhance general code generation capabilities. However, these methods largely rely on natural language descriptions (e.g., document strings, GitHub issues [54] or code review comments [24]) for the desired program logic. Even though some methods attempt to enhance the LLM's capability to understand intents expressed in natural language [24], ambiguity remains unavoidable: one cannot simply instruct an LLM to "avoid race conditions" and expect a correct outcome. Furthermore, these methods often include all project-related code in the context, requiring the Agent to autonomously decide how to retrieve information from this context. Such a burden is uncontrollable, and an excessively long context can potentially degrade the quality of code generation.

Insight and challenges. Our key insight is that we could guide the LLM using principles derived from formal methods, rather than imprecise natural language prompting. Such a formally structured specification is expected to address the challenges outlined in §1 simultaneously. First, a semantic gap arises when specifying module logic, necessitating resolution of (i) semantic ambiguities, (ii) deep domain knowledge awareness, and (iii) thread-safe requirements. Second, complex component composition demands careful consideration of inter-module dependencies and mitigation of cascading effects induced by each evolutionary patch across other modules. Third, the inherent unreliability of LLM capabilities poses a critical barrier to robust system generation.

3 Design Overview

This paper presents SYSSPEC, an end-to-end framework for developing specification-based generative file systems. SYSSPEC carefully applies principles from formal methods to create structured, high-level specifications. While not strictly formal, these specifications provide a sufficiently precise blueprint to effectively guide LLMs in generating and evolving complex FS implementations.

The SYSSPEC specification. At the core of our framework is a multi-part specification that captures a developer's design intent, as shown in Fig. 5-a. It consists of: (1) **Functionality specifications**, which use concepts like Hoare logic (pre/post-conditions) and invariants to describe the behavior of individual modules. (2) **Modularity specifications**, which decompose the system into distinct components and use a rely-guarantee discipline [21] to ensure they can be composed correctly and developed independently. (3) **Concurrency specifications**, which explicitly define locking protocols and other concurrency-related behaviors that are notoriously difficult for LLMs to infer on their own.

SYSSPEC shifts the developer's role from writing low-level C code to authoring a high-level design. This paradigm usually demands a greater upfront investment in design, much like the safety guarantees in Rust require more care than traditional C programming. However, the payoff is significant: once specified, the file system is more robust, easier to evolve, and better suited for diverse scenarios because the developer is forced to reason about the design concretely.

Besides, SYSSPEC streamlines evolution using **spec patches**, as shown in Fig. 5-c. Instead of manually modifying thousands of lines of C code to add a feature, a developer writes a patch containing either new specifications or modifications to an old one. Our toolchain then applies this patch and regenerates the low-level implementation, ensuring the

change is propagated correctly throughout the system.

The SysSpec toolchain. SysSpec provides an LLM-based toolchain that translates the high-level specification into an executable implementation. As illustrated in Fig. 5-b, this toolchain includes agents: The **SpecCompiler** agent translates the specification (SpecFS) into a low-level C-language implementation (`ImpFS`). To manage the current limitations of LLMs, it uses techniques like two-phase prompting to handle complex logic. The **SpecValidator** agent ensures the correctness of the generated code, employing a retry-with-feedback loop to automatically identify and correct errors. With the designs of SysSpec, our evaluation shows significant correctness improvements for complex operations compared to naive prompting (up to 34.4%). We also provide **SpecAssistant** to ease the development of specification.

Case study: SpecFS. With SysSpec, we design and implement SpecFS, a complete file system written entirely in the SysSpec specification language. Our toolchain can automatically generate a functional C-language implementation of SpecFS without human intervention. Through a series of case studies, we demonstrate that SpecFS can seamlessly evolve via spec patches to support sophisticated features found in Ext4, e.g., extents or delayed allocation.

4 SysSpec Framework

4.1 Functionality Specification

The functionality specification defines the behavior of a module by describing its state transitions. A module is a collection of related state variables and functions. The specification is built upon three components, inspired by formal methods.

First, **pre- and post-conditions**, following Hoare Logic, define the contractual obligations for each function by specifying the required state before execution and the guaranteed state upon completion. Second, **invariants** are properties that must hold true across all state transitions, ensuring the module's integrity. Such Hoare-logic-based specification approach inherently avoids the need for explicitly constructing the entire state space, thereby directly circumventing the problem of state space explosion. Finally, a **system algorithm** outlines the high-level logic for how a function should achieve its state transition, guiding the LLM's implementation strategy. Rather than always providing a complete system algorithm, our experience shows that a high-level **intent** is often sufficient. The intent can be regarded as a lightweight system algorithm that, expressed in natural language, guides the LLM in generating the desired implementation.

Not all of these components are required for every module. We find that the necessary level of detail scales with complexity. For straightforward modules (**Level 1**), pre/post-conditions and (sometimes) invariants are often sufficient. As the logic becomes more intricate (**Level 2**), adding an intent description is recommended to clarify the design. For the most complex cases involving highly optimized designs (**Level 3**), providing an explicit algorithmic description becomes essential, as it is unreasonable to expect an LLM to derive such logic from scratch.

```
1  /* Hoare-style Specification */
2  Pre-condition:
3    path: a NULL-terminated string array
4    name: a valid string
5  Post-condition:
6  Case 1 Successful traversal and insertion
7    - New inode created
8    - Entry inserted into target directory
9    - Return 0
10 Case 2 Traversal or insertion failure
11   Return -1
12 Invariant: root_inum always exists
```

Figure 6: Simplified functionality specificaiton for `atomfs_ins`

Hoare logic for file system synthesis. To specify function behavior, we adapt the classic Hoare logic formalism of {P}C{Q} using pre- and post-conditions, but we sidestep the high complexity of full formal verification. Each function is annotated with **pre-conditions** that define required system states and **post-conditions** that guarantee specific state transitions and return values. Fig. 6 shows `atomfs_ins`, an internal function of AtomFS [55] that implements `mknod`/`mkdir`. The pre-condition describes the validness of the parameters. The post-condition describes the state after the operation.

Our approach diverges from traditional formal methods in two key aspects. First, an LLM enforces adherence to this logic during code generation, replacing the role of a formal theorem prover. Second, to balance the comprehensibility and unambiguity of the specification, our specifications are expressed in structured natural language augmented with type annotations rather than pure mathematical logic, thereby making them more accessible. SysSpec augments the semantic precision of natural language with a structured organization for the specifications (e.g., sections for functionality, modularity, and concurrency), and uses mathematically disciplined natural-language expressions to ensure that the intended semantics remain precise and unambiguous while keeping the specification accessible. For example, the specification states that *the file size equals* $max(old\_size, offset + len)$, rather than that *the write updates the file size if necessary*. These designs significantly limit the potential for misinterpretation.

Invariant-guided specification. In addition to per-function contracts, developers specify system-wide **invariants** that describe properties valid across all states during execution. For example, an invariant may state that:

```
[Invariant] any modification of an inode must
occur while holding the corresponding lock
```

Such invariants define constraints that functions must respect throughout their execution, and cannot be fully expressed using only local pre- and post-conditions. For another example, the root existence invariant in Fig. 6 allows the generated code to safely omit null checks when accessing the root.

System algorithm. While pre- and post-conditions define

what a function must accomplish, they do not specify how. Our experience shows this is insufficient for performance-critical systems, as an LLM might generate an implementation that satisfies the specification but is highly inefficient. E.g., given a specification for a sort() function, an LLM could correctly generate a bubble sort ($O(n^2)$) just as easily as a quicksort ($O(n \log n)$). To address this, the **system algorithm** component allows developers to explicitly outline the method for achieving a state transition. This provides crucial guidance, ensuring the LLM implements a performant algorithm, such as using lock coupling for fine-grained concurrency instead of a coarse-grained global lock.

For example, in the functionality specification of `atomfs_rename`, we define the algorithm in three phases: (1) traversing the common path, (2) traversing the remaining path and (3) checks and operations. We explicitly specify the fine-grained locking scheme used in these phrases, which enables the correct generation of `atomfs_rename`, a function that is both highly complex and prone to deadlock.

Intent. LLMs are trained on vast codebases and can often produce highly optimized code when given the right high-level guidance, even without a complete system algorithm. The **intent** component in SysSpec is designed to provide such guidance. First, it describes the high-level goal of a function in natural language, e.g., "successful traversal and insertion" in Fig. 6 indicates that the target directory is identified through file-tree traversal. Second, it allows developers to inject domain-specific knowledge to steer the LLM toward better implementation choices, complementing the correctness guarantees provided by the Hoare logic and invariants. E.g., when reading a large file extent, a developer can use the intent to suggest a single, bulk I/O operation. Without this hint, an LLM might generate a correct but inefficient implementation that reads each disk block individually.

4.2 Modularity Specification

SysSpec addresses the composition challenge, avoiding interface-level dependency errors through a methodology that combines interface contracts with LLM context management.

Context-bounded modular synthesis. One of the core innovations in SysSpec's modularity specification lies in aligning module design with two intrinsic properties of LLM reasoning. First, strict size constraints ensure each module fits entirely within the model's context window, enabling holistic analysis during generation. The specific constraints on module sizes evolve in tandem with improvements in LLM capabilities and context window capacities. Take our case study (§5) as an example, we limited module sizes to $\leq$500 LoC, which keeps the token consumption for inference generally within approximately 30K tokens. Second, explicit interface contracts govern inter-module dependencies through *Rely-Guarantee* conditions, a formal mechanism adapted from concurrent program verification [55]. This methodology embodies three critical principles: (1) module implementations must respect their declared dependencies (Rely), (2) provide guarantees about their behavior (Guarantee), and (3) compose through logical implication of these contracts.

```
[RELY]
Predefined Structures/Functions:
struct inode { ... };
struct inode* root_inum;
void lock(struct inode*)
void unlock(struct inode*)
struct inode* locate(struct inode* cur, char*
    ↪ path[]) // Traverse path under cur
void insert(struct inode*, struct inode*, char*)
    ↪ // ...
int check_ins(struct inode*, char*) // ...

[GUARANTEE]
Exported Interface:
int atomfs_ins(char*[], char*, int,
               unsigned, unsigned)
```

Figure 7: **Rely-Guarantee specifications for `atomfs_ins`.** Both rely and guarantee conditions are simplified for clarity.

A critical adaptation occurs in re-imagining rely-guarantee reasoning, originally developed for concurrent thread verification, for modular system synthesis. Where traditional *rely* conditions specify permissible environment interference for threads, SysSpec's *Rely* clauses enumerate a module's assumptions about other components. Similarly, *Guarantee* clauses replace thread behavior specifications with module interface contracts. Each module's Rely conditions must be entailed by the Guarantees of its dependencies, enabling compositional correctness through localized synthesis.

In Fig. 7, the module's Rely clause imports critical elements from its dependent modules, e.g., the definition of structures, lock/unlock primitives, path traversal function. The generated code then exports its own Guarantee, allowing dependents to build upon its functionality without needing to understand internal implementation details.

Incorporation with external code. SysSpec also supports incorporating external code (e.g., libraries) via the Rely-Guarantee framework. External code can be integrated by first exposing their *Guarantees*. Developers then specify dependencies on these exposed guarantees within the *Rely* clause of their specifications. During code generation, the external code is treated as a satisfied dependency, enabling the generated module to correctly invoke external functions without reimplementing them.

4.3 Concurrency Specification

Concurrency poses one of the most significant challenges in FS implementation. A naive approach within the SysSpec would be to describe concurrent behavior using the existing functionality specification, defining lock states as pre- and post-conditions and outlining the logic in the algorithmic description. While plausible in theory, our experience reveals that LLMs struggle to correctly synthesize complex concurrent logic from such unified specifications alone. E.g., when tasked with implementing the notoriously complex rename, state-of-the-art LLMs consistently failed to generate a correct

```
[Rely]
[Locking Specifications of locate]
Pre-condition: cur is locked.
Post-condition: suppose return target.
- if target is NULL, no lock owned.
- if target is not NULL, only target is owned.
[Locking Specifications of check_ins]
Pre-condition: cur is locked.
Post-condition:
- if check_ins returns 0, cur is locked.
- if check_ins returns 1, no lock is owned.

[Locking Specifications of atomfs_ins]
Pre-condition: no lock is owned.
Post-condition: no lock is owned.
```

Figure 8: **Concurrency specifications for `atomfs_ins`.**

```c
int atomfs_ins(char* path[], char* name, ...) {
    lock(root_inum);
    struct inode* target = locate(root_inum, path);
    if (!target) return -1;
    if (check_ins(target, name) != 0) return -1;
    struct inode* new_inode = malloc_inode(...);
    insert(target, new_inode, name);
    unlock(target);
    return 0;
}
```

Figure 9: **LLM-generated `atomfs_ins` with SysSpec.**

implementation that satisfied the specification.

Our key insight is to decouple concurrent logic from the functional logic. We separate the concurrency design (e.g., locking) into a standalone **concurrency specification**. This specification is a specialized version of the functionality specification, focusing solely on concurrent behavior.

During code generation, our toolchain first directs the LLM to generate a correct *sequential* version of the code, focusing only on the primary functionality. Once this sequential implementation is validated, the toolchain performs a second pass, using the dedicated concurrency specification to instrument the code with the required locking and other concurrent behaviors. This separation of concerns makes the synthesis task tractable for current LLMs, allowing them to correctly handle two distinct, complex problems one at a time.

Fig. 8 shows an example of the concurrency specification. The implementation of `atomfs_ins` relies on `locate` and `check_ins`. In its concurrency specification, the **Rely** clauses capture the locking requirements of these internal functions. Specifically, since `locate` requires the `cur` lock as a precondition, while `atomfs_ins` itself has no such precondition, the generated code must first acquire the lock on `root_inum` before invoking `locate(root_inum, path)`. This implies that when implementing a module with SysSpec, the LLMs considered not only the functional dependencies between modules but also their locking dependencies.

Putting it together. Fig. 9 shows an example of generated `atomfs_ins`. The code fulfils the functionalities defined by the pre- and post-conditions, correctly invokes functions from other modules in accordance with the modularity specification, and handles lock acquisitions/releases properly as specified by the concurrency specification.

4.4 DAG-Structured Specification Patch

We introduce a DAG (Directed Acyclic Graph) structured specification patch, a mechanism that simplifies the evolution of spec-based file systems. This approach provides a **self-contained** description of a new feature, explicitly organizes the dependency changes, and defines a clear and consistent workflow for applying the evolution.

The leaf node: a self-contained change. The evolution process begins at a **leaf node** of the DAG, which has no dependencies on other patch nodes. The specification in a leaf node represents a localized, self-contained change, typically within a single module. It introduces new logic and provides new guarantees without affecting any other part of the existing system. A leaf node can also define new data structures or functions that subsequent nodes in the patch will rely on.

Intermediate nodes: building on guarantees. An intermediate node represents a step in the evolution that builds upon previously introduced changes. Its specification relies on the new guarantees provided by its child nodes to implement more complex logic. In turn, this node provides its own, higher-level guarantees, forming a clear dependency chain that progressively constructs the new feature. In principle, any modification to a guarantee necessitates corresponding specification adjustments for all modules that depend on it, i.e., these modules are included in the patch as intermediate nodes to preclude potential compatibility issues.

The root node: the integration point. The root node is the culmination of the evolution, acting as the final integration point. Unlike the "tree" structure, a "DAG" structured patch may have multiple root nodes. Root nodes are characterized by the property that their specification provides semantically unchanged guarantees. This equivalence is critical, as it allows the entire chain of new functionality, built up through the DAG, to safely and transparently replace an old implementation. This substitution serves as the **"commit point"**, where the evolution is atomically applied to the base system.

Evolution involving existing modules. Nodes within a DAG-structured specification patch includes both new modules and modifications to existing ones. A modified existing module is treated as a "new module," which can largely reuse the existing specification. If a shared component (e.g., `inode`) is modified, all dependent modules must be regenerated to "rely" on the updated version, while other parts of their specifications may not require any modifications. These modified modules then replace the original ones after the patch is merged.

The evolution process. The unidirectional dependencies between nodes naturally form a DAG, which dictates the evolution workflow. The process begins with the SysSpec toolchain (§4.5) generating code for the leaf nodes. It then traverses the graph upwards, synthesizing the implementation for each parent node by leveraging the freshly generated guarantees from its children. This continues until the toolchain reaches the root node, at which point the new feature is fully

implemented and integrated into the file system.

4.5 The SysSpec Toolchain

We have designed and implemented three LLM-based agents, SpecCompiler, SpecValidator, and SpecAssistant, that form the core toolchain for the SysSpec to serve the entire lifecycle of generative file system development, The toolchain further effectively mitigates hallucinations in LLMs by addressing two key aspects: (1) it helps that LLMs produce outputs aligned with those specifications through the coordination of SpecCompiler and SpecValidator, and (2) it helps developers in producing correct specifications by leveraging a strictly structured specification format and the SpecAssistant.

The SpecCompiler agent. The SpecCompiler agent is responsible for translating the high-level, specification-based file system into a low-level C-language implementation (`ImpFS`) that can be compiled and deployed. Analogous to a traditional compiler, it processes the source specification and outputs machine-usable code. Thanks to our modular design, the SpecCompiler can operate on one module at a time, confident that the strict enforcement of rely-guarantee conditions will ensure seamless integration into a monolithic whole.

For each module, the SpecCompiler employs two primary techniques. The first is **two-phase prompting**, which leverages our separation of concerns in the specification. The agent's first phase generates a correct sequential implementation of the module, focusing only on its core functionality. In the second phase, it uses the dedicated concurrency specification to instrument this sequential code with the necessary locking and concurrent behaviors.

The second technique is an iterative **retry-with-feedback** loop used within each phase. This loop involves two distinct LLM roles: a `CodeGen` agent generates the implementation, and a separate, reasoning-focused `SpecEval` agent reviews the output against the specification. If the `SpecEval` agent identifies a flaw, it does not simply report failure; instead, it generates specific, actionable feedback (e.g., "The case where function foo() fails is not handled"). This feedback is then appended to the original prompt, and the `CodeGen` agent retries. This refinement cycle continues until the generated code satisfies the specification or an attempt-limit is reached.

This dual-agent design is critical for overcoming LLM hallucination. In our experience, the code-generation LLM will occasionally produce incorrect implementations, even with a precise specification. However, the `SpecEval` agent effectively detects these flaws. This is because verifying a solution against a set of rules is a simpler cognitive task than generating the solution from scratch, and the probability of two distinct models making complementary errors on the same logic is exceedingly low.

The SpecValidator agent. The SpecValidator agent performs the final, holistic verification of the complete `ImpFS`. It combines specification-based review with traditional testing. First, it re-uses `SpecEval` logic from SpecCompiler to check each fully-generated module against the combined functionality and concurrency specifications. Second, it integrates with standard software engineering workflows by running a suite of unit and regression tests against the final C-language file system. This process emulates a modern CI/CD pipeline: the `SpecEval` component acts as an automated code reviewer verifying adherence to the design, while the test suite ensures that no existing functionality has regressed.

The SpecAssistant agent. SpecAssistant streamlines specification development through a human-in-the-loop process. A developer provides a draft specification, which the SpecAssistant first validates and reformats to meet SysSpec's syntax. The agent then enters an automated refinement loop. It repeatedly invokes the SpecCompiler; if the SpecCompiler's `SpecEval` phase identifies a flaw, the SpecAssistant uses a new `SpecFine` step to automatically polish the specification based on the feedback before retrying. This loop concludes in two ways. On success, the SpecAssistant provides the developer with the refined specification and the C implementation for validation. On failure, it returns the last attempted specification annotated with detailed diagnostics, serving as a debug log that guides the developer in resolving the issue.

5 SpecFS: A Case for Generative FS

5.1 Prototype Implementation

Overview. To demonstrate the benefits of SysSpec, we introduce SpecFS, a concurrent in-memory file system that runs in userspace via FUSE. The architecture is based on a prior formally verified file system, AtomFS [55]. However, rather than porting its C implementation, we undertake a *complete reimplementation* guided by our specification. Specifically, we implement AtomFS's high-level design with SysSpec's specifications, including pre- and post-conditions (we re-use some conditions from AtomFS's formal specifications in Coq), invariants, rely/guarantee contracts, system algorithm descriptions, and concurrency specification.

SpecFS is organized into 45 distinct modules distributed across several logical layers, including file operations, inode management, path traversal, and the POSIX interface. SpecFS supports a wide range of standard POSIX calls, such as open, read, and rename. To validate its functional correctness, we use the `xfstests` [17] suite within our SpecValidator. SpecFS demonstrates a level of correctness equivalent to that of the original AtomFS, failing only 64 out of 754 test cases, all attributable to unimplemented functionality.

System complexity and comprehensibility. While SpecFS serves as our prototype, it is a non-trivial file system by any measure. The generated C implementation comprises approximately 4,300 lines of code. To put this figure into context, we compared it against the 82 file systems in the Linux 6.1.10 kernel. SpecFS ranks 42nd by line count, surpassing established systems like `squashfs` and nearly matching the size

Table 2: **Case study of applying Ext4 features to AtomFS**. *"Propose" and "Launch" indicate the year an feature proposed and the year it is merged. "Release" indicates the Linux version with the feature. "Indirect Block" is an exception in the table, as it originates from ext2/3.*

Type	Feature	Propose	Launch	Brief Description	Release
I	Indirect Block (Ext2/3)	/	/	One-to-one block mapping via multi-level pointers	/
	Extent	2006	2006	Contiguous block ranges reducing metadata by 50%	2.6.19
	Inline Data	2011	2013	Store small files in inode's unused space	3.8
II	Multi Block Pre-Allocation	2006	2008	Benefit large files by allocating blocks in groups	2.6.25
	Delayed Allocation	2006	2008	Deferred block allocation for reducing I/O operations	2.6.27
	rbtree for Pre-Allocation	2022	2023	rbtree to organize the pre-allocated block pool	6.4
III	Metadata Checksums	2011	2012	Checksummed filesystem metadata structures	3.5
	Encryption	2015	2015	Per-directory encryption with low overhead	4.1
	Logging(jbd2)	2006	2006	Journaling support for 64-bit filesystems	2.6.19
IV	Timestamps	2006	2006	Nanosecond resolution timestamps in inode structure	2.6.19

of `9pfs`. We choose C as the programming language as it remains the de facto standard and is most widely used for low-level file system developments. While high-level languages like Rust may be applicable in the future, they present trade-offs between the benefits from their inherent features (e.g., memory safety) and the increased specification complexity (e.g., due to Rust's ownership model). Moreover, we observe that the generated code of SPECFS is highly comprehensible. LLMs naturally adhere to standard coding conventions and generate explaining comments, mirroring the style of human engineers.

Workflow. SPECFS features a unique runtime translation workflow, as shown in Fig. 5-b. It begins with our LLM-based agents generating C code for each module based on its specification. A background daemon then compiles the generated code and validates it using the SpecValidator. Once validated, the compiled components are deployed into the host OS. To mitigate the latency associated with LLM-based generation, successfully validated module implementations are cached for immediate reuse. When the specification is updated, regeneration is triggered asynchronously, allowing the file system to remain fully operational while the new version is prepared in the background.

Experience for developing SPECFS. We gain several experiences during the development of SPECFS. First, while strict SYSSPEC adherence effectively ensures accuracy of code generation (§6.1), simplified specifications (e.g., simplify Rely/Guarantee) can occasionally still result in logically correct code. If the objective is to maximize development efficiency, one might consider making a trade-off between generation accuracy and the use of simplified specifications. Second, to debug SPECFS, we initially locate bugs in the generated C code, consistent with debugging methods for human-written code. Once a bug is localized, the debugging process additionally involves validating the high-level design against the specification. In our experience, most issues in SPECFS are diagnosed at the specification level.

5.2 Case Study of Evolutions

One benefit of a specification-based generative FS is the enhancement of the evolvability. In this section, we evolve the design of SPECFS with 10 well-known Ext4's features (Tab.2) using spec patches[1], as a case study. These ten features are classified into four categories that current SYSSPEC can support: (I) *File structure modification*, which modifies the underlying data structures within the file system; (II) *Design update for existing operations*, which modifies the specific behavior of existing operations, without altering the intended outcome of these operations; (III) *New functionalities implementation* with new operations; (IV) *Hyperparameters or metadata modification*, e.g., changing the FS's block size from 32-bit to 48-bit. Tab.2 presents the specific type of each feature and a brief description of its implementation logic.

Example: Extent. This feature transforms the file structure from individual blocks into `extents`. Each extent records a segment of contiguous blocks, facilitating sequential reads and writes. The modification process (Fig. 10) begins by defining new data structure for `extent` and `inode`. Then, it updates the low-level file operations in `lowlevel_file` (and corresponding initializations) to incorporate new extent-based file I/O operations. Subsequently, the `inode_management` module is modified to invoke these new extent operations. Since the new `inode_management` provides the same guarantee as the original `inode_management`, the new `inode_management` serves as the root node of the entire patch and directly replaces the original one, making the complete set of new operations visible to the entire system.

6 Evaluation

6.1 Evaluations on Accuracy

Methodology. We first study whether SPECFS can accurately generate code to achieve self-evolution, that is, whether the

[1]The Appendix supplements the DAG structures of the patches used for specifying these features.

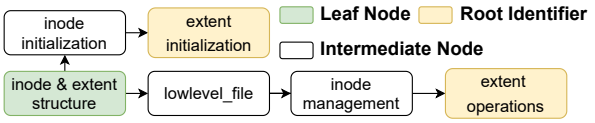
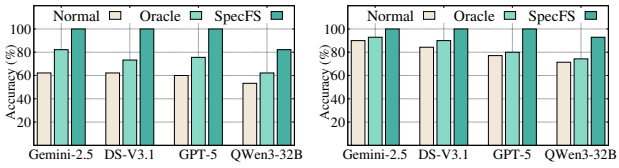


Figure 10: **DAG-Structured patch for implementing "Extent".** *For simplicity, a node in the figure may encompass multiple modules of specifications. "Root Identifier" indicates that this node is the root (along with its associated logic).*



(a) AtomFS. **(b) Features.**

Figure 11: **Accuracy results for implementing AtomFS or new features.** *"GPT-5" in the figure denotes GPT-5-minimal.*

code generation can accurately conform to the logic described by the specification and accurately meet the functionality of the corresponding file system module. Our test cases comprise all the modules to implement the complete logic of AtomFS [55]. Specifically, to evaluate the accuracy, we first define 45 distinct modules within AtomFS and manually author their ground-truth implementations. A generated module is considered **correct** if it (i) passes all functional tests for AtomFS and (ii) is deemed logically equivalent to the ground-truth through manual inspection.

We implement two versions of baselines based on a few-shot learning approach. The normal version incorporates a description of the file correspondence logic and the APIs of the dependency modules; The oracle version not only includes the dependency module's APIs but also integrates the ground-truth code of these modules as part of the prompt. We test generation accuracy using four LLMs of decreasing capability: Gemini-2.5-Pro [7], DeepSeek-V3.1 Reasoning [18], GPT-5-minimal [8], and Qwen3-32B [51], whose performance is ranked according to the LiveCodeBench leaderboard [9].

Accuracy results. To evaluate accuracy, SPECFS uses the same specification to generate code for the 45 modules with the four models, and the accuracy results are presented in Fig. 11-a. We observe that on more powerful models such as Gemini-2.5-Pro and DS-V3.1, SPECFS achieves 100% accuracy, completely generating all functional modules of SPECFS. In contrast, even when the oracle baseline implementation possesses all contextual code, the accuracy of code generation using the most capable Gemini-2.5-Pro is 81.8%. This demonstrates that SPECFS's specification design significantly enhances code generation accuracy for the file system.

6.2 Evaluations on Generalizability

We further evaluate how SYSSPEC can support the generation of a wider range of file system logic. To this end, we evaluate whether SPECFS can accurately generate the ten features detailed in Tab. 2. These ten new features encompass a total of 64 functional modules, and we report their overall accuracy. As depicted in Fig. 11-b, SPECFS consistently exhibits higher generation accuracy across all evaluated models. This further substantiates the effectiveness of SYSSPEC. Noted that, due to many features being implemented primarily through modifications to existing specifications, and involving less complex concurrency logic, the accuracy of implementing features is higher than implementing atomfs from scratch, further reflecting the feasibility of using LLMs for FS evolving.

Table 3: **Ablation study.** *The evaluation uses DeepSeek-V3.1 Reasoning. "Func", "Mod", "Con" means Functionality, Modularity and Concurrency Specifications respectively.*

Modules	Func	+Mod	+Con	+SpecValidator
Concurrency-agnostic	40.00% (12/40)	100% (40/40)	100% (40/40)	100% (40/40)
Thread-safe	0% (0/5)	0% (0/5)	80% (4/5)	100% (5/5)

Generalizability for various locking methods. We evaluate whether the concurrency specifications of SYSSPEC are general for various locking methods. To this end, we utilize the `dentry_lookup` operation in the VFS layer of Linux, as it exhibits locking requirements at multiple granularities: a lock must be acquired for the entire hash list during traversal, while individual locks are also required for each dentry upon access. In our concurrency specification, we explicitly designate the use of two distinct locking mechanisms—lock-free RCU for the hash list and spinlocks for individual dentries. Our experiments verify that the generated code correctly adheres to the specified concurrency semantics, successfully producing code that implements multi-granularity and multi-method locking logic as intended. See details about the specification and the generated code of the two phases in the Appendix.

6.3 Ablation Study

We evaluate how SPECFS's designs effectively improve the accuracy of code generation. We first divide AtomFS's 45 modules into 40 concurrency-agnostic modules and 5 thread-safe modules, and then assess the accuracy for implementing these modules under different design configurations, as shown in Tab. 3. According to the evaluation, we observe that although the functionality specification alone is insufficient for correctly generating complex file system (primarily due to interface mismatch), when combined with the modularity specification, it effectively supports the generation of concurrency-agnostic modules. Nevertheless, that is not enough to accurately produce thread-safe modules. SPECFS further addresses this limitation by incorporating a concurrency specification and agent-supported self-validation to achieve correctness and robustness for thread-safe modules.

6.4 Evaluations on Productivity

We further evaluate how SPECFS improves the productivity. We compare the development costs of manual implementa-

Table 4: **Productivity improvement.**

Development Costs	Extent	Rename
Manual	4.5h (3.0×)	13h (5.4×)
Ours	1.5h	2.4h

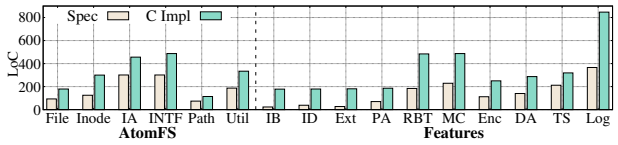


Figure 12: **Lines of code comparison between specification and source code.** *The results contain six basic logical layers of AtomFS and ten new features in Tab. 2. The abbreviations are as follows. IA: Interface Auxiliary; INTF: Interface; Util: Utility; IB: Indirect Block; ID: Inline Data; Ext: Extent; PA: Pre-Allocation; RBT: rbtree for Pre-Allocation; MC: Metadata Checksums; Enc: Encryption; DA: Delayed Allocation; TS: Timestamps; Log: Logging.*

tion and specification-driven generation for the following two patches: (1) Supporting the "extent" feature for original AtomFS, which requires updating multiple concurrency-agnostic modules; and (2) Implementing the "rename" module of AtomFS, which involves complex locking logic to ensure thread-safety. We invite two CS master students and two PhD students for the evaluation.

As shown in Tab. 4, SPECFS improves the programming productivity of the two implementations by 3.0× and 5.4×. For implementing patches involving multiple modules, SPECFS's design of DAG-structured specification patch allows for faster identification of all modules requiring modification, without the need for source code analysis. SPECFS's concurrency specifications further reduce the complexity of developing sophisticated thread-safe functions.

Lines-of-code. We conducted a comparative analysis of the lines of code (LoC) between the specifications and the corresponding generated C source code. The specifications of AtomFS are categorized by logical layers, each of which may encompass several modules. The specifications of Features are categorized based on their functional characteristics, each of which corresponds to the features listed in Tab. 2. The results in Fig. 12 show that the specification descriptions consistently require fewer lines than their corresponding generated C source code. This reduction in LoC implies possibly lower development effort and improved productivity.

Generation latency. The latency of code generation primarily depends on the inference performance of the LLM itself. In our experience, the generation time for SPECFS typically ranges from several minutes to tens of minutes.

6.5 Performance Optimizations

This section evaluates how the new features implemented by SPECFS (§5.2, Tab. 2) effectively improve the performance.

Inline data. We first evaluate the compression effect of "Inline Data" on file sizes. As shown in Fig. 13-left, using inline data significantly reduces the file sizes of QEMU [10] and Linux source code [6], decreasing the required storage capacity by 35.4% and 21.0%, respectively.

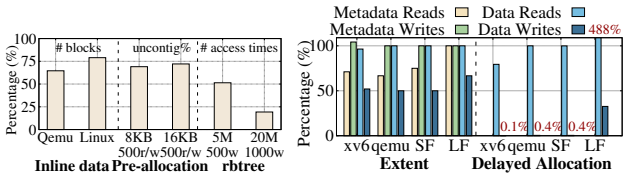


Figure 13: **Performances improvements with new features implemented by SPECFS.** *The results were normalized before and after optimization, and a lower percentage indicates better performance. Test cases in the right are "xv6 compilation" ("xv6"), "copy qemu" ("qemu"),"small file" ("SF"), "large file" ("LF"). The "SF" and "LF" tests involve various read or write operations on multiple small or large files, respectively representing metadata-intensive and data-intensive workloads. The data for "Delayed Allocation" that is either too long or too short is marked in the figure.*

Multi-block pre-allocation. We evaluate whether the "Multi-block Pre-allocation" feature, which integrates "Extent" and proactively allocates contiguous blocks to form a block pool and subsequently prioritizes drawing blocks from this pool during block allocation, increases the sequential read/write ratio of file system operations. Our microbenchmarks first create a large file and issue random writes to it at fixed page sizes (e.g., 4KB or 8KB). Then, we repeatedly apply the following process: a random region within the file is selected, and sequential read or write operations are issued over this range. Fig. 13-left shows uncontiguous reads and writes ratio drops ∼30% after applying pre-allocation (we regard a read-/write operation as sequential if its range falls within a single extent). This emphasizes that our pre-allocation optimization enhances the contiguity of data blocks of each file.

Red-black tree for pre-allocation. We evaluate how effectively the transition of the pre-allocation block pool from a linked list to a red-black tree structure reduces block pool access frequency. Our micro-benchmark first constructs a file with a large block pool by employing a series of write operations exhibiting a specific pattern. Then, we perform random writes to the file and record the number of accesses to the block pool. As shown in Fig. 13-left, the red-black tree demonstrably reduces block pool access frequency, for example, by 80.7% when performing 1,000 writes on a 20MB file. The result also indicates that the benefits of the red-black tree are more pronounced with larger files.

Extent. Fig. 13-right shows the proportion of I/O operations (encompassing both metadata and data reads and writes) after applying Extent, relative to those before applying Extent. The result shows that applying Extent effectively reduces the number of I/O operations, thereby improving the performance of the file system. It is because an extent data structure represents a sequence of contiguous data blocks. The read and write operations on this sequence of data blocks are completed in a single I/O operation, rather than through multiple individual block-by-block reads and writes.

Delayed allocation. We evaluate how the number of data

read and write operations is reduced by "Delayed Allocation", which prioritizes read and write within a global buffer, and flushes the buffer to the disk in a batch when it reaches the size limits. According to the results shown in Fig. 13-right, the number of write operations is significantly reduced, with some cases demonstrating an elimination of up to 99.9% of write operations. Read operations are also reduced in most cases, while in the cases such as the large file test, the number of data reads increases. The reason is that, after applying the feature, data writes no longer directly target the disk; instead, data is read into a buffer and write operations are performed within that buffer. This may introduce additional reads, especially for regular sequential cyclic writes.

6.6 Limitations and Discussion

Missing FS features. Although SPECFS represents a successful endeavor in realizing a complex file system using SYSSPEC, SPECFS is currently implemented as a user-space file system based on FUSE [5]. Consequently, it does not operate in kernel mode and lacks a storage stack, such as direct disk access, nor does it consider crash consistency. Our evaluation of SPECFS primarily focuses on validating the generation methodology and ensuring correctness, which precludes an apple-to-apple comparison with native kernel-space file systems regarding raw performance metrics such as throughput.

We plan to apply SPECFS to the self-evolution of industrial file systems (e.g., EROFS or Ext4 [4, 23]), which is challenging due to the increased engineering complexity. To address this, we consider a fully formally verified "AtomFS-Ext4" unnecessary. Instead, we propose developing a "SPECFS-Ext4" directly based on documentation, using these mitigation strategies. First, instead of specifying the whole state of Ext4, developers can start with a minimal baseline (like Ext2) and incrementally add features to it. Second, we plan to enhance the SpecAssistant to automatically bootstrap draft specifications from documentation (e.g., kernel wikis) or even Ext4 source code. Third, we could adapt methodologies of software engineering to guide the development of specification-constructed (rather than programming-language-constructed) file systems, e.g., applying methods similar to the Law of Demeter [32] to reduce the coupling between modules.

Push-button verification integration. Moreover, although SpecValidator enhances the correctness of generated code to some extent through software testing and LLM-based validation, a greater potential of SPECFS lies in the fact that each module is equipped with a ready specification. This inherently facilitates integration with push-button verification and similar methodologies, holding the potential to achieve a generative and formally-verified paradigm.

7 Related Work

We present other related work besides the discussion in §2.3.

Domain-specific code generation. Some efforts in program synthesis (e.g., SyGus [39]) also focus on automated code generation. They may leverage Domain-Specific Languages (DSLs) to precisely articulate code logic, but face the challenges in scaling to diverse types of scenarios. E.g., MegaLibm [12] introduces a novel DSL to construct specifications for mathematical library functions. DryadSynth [19] focuses on bit-vector synthesis building upon the SyGus methodology.

Other research addresses LLM-based code generation for specific domains outside file systems. QiMeng Xpiler [20] uses LLMs to construct a transcompiler that adapts low-level tensor programs for various hardware platforms. Autoverus [52] employs LLM-based tools for proving the correctness of Rust code. OSVBench and SpecGen [30, 37] address generating formal specifications for a given code snippet.

Formal verification methods. Many studies verify complex systems, such as operating systems [28], file systems [16, 55, 56], or cloud systems [47]. These efforts also involve formal specifications to ascertain the correctness of handcrafted implementations, but cannot be used diretly for generative file systems. We can draw upon these existing specifications when authoring SYSSPEC for code generation.

LLM-assisted development. A separate stream of research applies LLMs to mitigate the burden of the file system development. For example, WASABI [44] utilizes LLMs to detect intricate retry-related problems in large-scale systems. SysGPT [40] employs LLMs to provide developers with context-aware performance suggestions. However, they cannot effectively evolve a file system like SPECFS.

8 Conclusion

This paper presents SYSSPEC and SPECFS, a framework and a case for generative file systems. Different from traditional paradigms, SYSSPEC shifts the developer's focus from writing low-level code to designing high-level specification, and SPECFS shows the potential benefits on evolvability.

Acknowledgments

We are grateful to our shepherd Mai Zheng for his detailed suggestions, which significantly improved the paper. We thank the anonymous FAST reviewers for their constructive feedback. This work was supported in part by the National Natural Science Foundation of China (No. 62432010, 62302300 and 62472279), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (JYB2025XDXM113), and the Fundamental Research Funds for the Central Universities. Corresponding authors: Dong Du (dd_nirvana@sjtu.edu.cn), Yubin Xia (xiayubin@sjtu.edu.cn), and Haibo Chen (haibochen@sjtu.edu.cn).

References

[1] Openai chatgpt application. https://openai.com/chatgpt/overview/, 2024.

[2] Copilot: Your ai companion. https://copilot.microsoft.com/, 2025. Referenced May 2025.

[3] Cursor. https://cursor.com/, 2025. Referenced May 2025.

[4] ext4(5) — linux manual page. https://man7.org/linux/man-pages/man5/ext4.5.html, 2025.

[5] Fuse. https://www.kernel.org/doc/html/next/filesystems/fuse.html, 2025.

[6] Github - torvalds/linux: Linux kernel source tree. https://github.com/torvalds/linux, 2025.

[7] Google gemini 2.5 application. https://gemini.google.com/app, 2025.

[8] Gpt-5 system card. https://cdn.openai.com/gpt-5-system-card.pdf, 2025.

[9] Livecodebench benchmark leaderboard. https://artificialanalysis.ai/evaluations/livecodebench, 2025.

[10] Qemu: A generic and open source machine emulator and virtualizer. https://www.qemu.org/, 2025. Referenced May 2025.

[11] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. Qwen technical report. *arXiv preprint arXiv:2309.16609*, 2023.

[12] Ian Briggs, Yash Lad, and Pavel Panchekha. Implementation and synthesis of math library functions. *Proc. ACM Program. Lang.*, 8(POPL), January 2024.

[13] William Bugden and Ayman Alahmar. Rust: The programming language for safety and performance. *arXiv preprint arXiv:2206.05503*, 2022.

[14] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

[15] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 18–37. ACM, 2015.

[16] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[17] Jonathan Corbet. Toward better testing. https://lwn.net/Articles/591985/, 2014.

[18] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan,

Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2025.

[19] Yuantian Ding and Xiaokang Qiu. Enhanced enumeration techniques for syntax-guided synthesis of bit-vector manipulations. *Proc. ACM Program. Lang.*, 8(POPL), January 2024.

[20] Shouyang Dong, Yuanbo Wen, Jun Bi, Di Huang, Jiaming Guo, Jianxing Xu, Ruibai Xu, Xinkai Song, Yifan Hao, Xuehai Zhou, et al. Qimeng-xpiler: Transcompiling tensor programs for deep learning systems with a neural-symbolic approach. *arXiv preprint arXiv:2505.02146*, 2025.

[21] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, page 315–327, New York, NY, USA, 2009. Association for Computing Machinery.

[22] Xiang Gao, Mingkai Dong, Xie Miao, Wei Du, Chao Yu, and Haibo Chen. EROFS: A compression-friendly readonly file system for resource-scarce devices. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 149–162, Renton, WA, July 2019. USENIX Association.

[23] Xiang Gao, Mingkai Dong, Xie Miao, Wei Du, Chao Yu, and Haibo Chen. EROFS: A compression-friendly readonly file system for resource-scarce devices. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 149–162, Renton, WA, July 2019. USENIX Association.

[24] Qi Guo, Xiaofei Xie, Shangqing Liu, Ming Hu, Xiaohong Li, and Lei Bu. Intention is all you need: Refining your code from your intention. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pages 1127–1139. IEEE Press, 2025.

[25] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2024.

[26] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Trans. Inf. Syst.*, 43(2), January 2025.

[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Safe systems programming in rust: The promise and the challenge, 2021.

[28] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[29] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[30] Shangyu Li, Juyong Jiang, Tiancheng Zhao, and Jiasi Shen. Osvbench: Benchmarking llms on specification generation tasks for operating system verification. *arXiv preprint arXiv:2504.20964*, 2025.

[31] Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 455–468, 2012.

[32] Karl J. Lieberherr and Ian M. Holland. Assuring good style for object-oriented programs. *IEEE Softw.*, 6(5):38–48, September 1989.

[33] Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

[34] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[35] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 31–44, San Jose, CA, February 2013. USENIX Association.

[36] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. *ACM Trans. Storage*, 10(1), January 2014.

[37] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. Specgen: Automated generation of formal program specifications via large language models. *arXiv preprint arXiv:2401.08807*, 2025.

[38] Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. A Concurrent Specification of POSIX File Systems. In Todd Millstein, editor, *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:28, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[39] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The sygus language standard version 2.1. *arXiv preprint arXiv:2312.06001*, 2023.

[40] Sujin Park, Mingyu Guan, Xiang Cheng, and Taesoo Kim. Principles and methodologies for serial performance optimization. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 357–373, 2025.

[41] Harshad Shirwadkar, Saurabh Kadekodi, and Theodore Tso. FastCommit: resource-efficient, performant and cost-effective file system journaling. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 157–171, Santa Clara, CA, July 2024. USENIX Association.

[42] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. Push-Button verification of file systems via crash refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 1–16, Savannah, GA, November 2016. USENIX Association.

[43] Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. Api-driven program synthesis for testing static typing implementations. *Proc. ACM Program. Lang.*, 8(POPL), January 2024.

[44] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. If at first you don't succeed, try, try, again...? insights and llm-informed tooling for detecting retry bugs in software systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pages 63–78, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Ion Stoica, Matei Zaharia, Joseph Gonzalez, Ken Goldberg, Koushik Sen, Hao Zhang, Anastasios Angelopoulos, Shishir G. Patil, Lingjiao Chen, Wei-Lin Chiang, and Jared Q. Davis. Specifications: The missing link to making the development of llm systems an engineering discipline. *arXiv preprint arXiv:2412.05299*, 2024.

[46] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. *arXiv preprint arXiv:2310.17807*, 2024.

[47] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666, Santa Clara, CA, July 2024. USENIX Association.

[48] Gemini Team, Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[49] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*, 2025.

[50] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817*, 2025.

[51] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[52] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. Autoverus: Automated proof generation for rust code. *Proc. ACM Program. Lang.*, 9(OOPSLA2), October 2025.

[53] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024.

[54] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, pages 1592–1604, New York, NY, USA, 2024. Association for Computing Machinery.

[55] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. Using concurrent relational logic with helpers for verifying the atomfs file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 259–274, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Mo Zou, Dong Du, Mingkai Dong, and Haibo Chen. Using dynamically layered definite releases for verifying the RefFS file system. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 629–648, Santa Clara, CA, July 2024. USENIX Association.

A Artifact Appendix

Abstract

This artifact contains the source code and scripts required to reproduce the results presented in the paper. The artifact includes the SpecFS filesystem generation pipeline using Large Language Models (LLMs) and end-to-end evaluation workflows. It supports reproducing the filesystem generation from high-level specifications, executing automated accuracy and performance benchmarks, and regenerating the plots reported in the paper.

Scope

The artifact allows for the validation of the following main claims made in the paper:

- **Accuracy:** The SpecFS filesystem generated by the framework accurately implements the given specifications. Validation is confirmed when the generated filesystem passes all functional tests in the pipeline.
- **Productivity:** The specification descriptions consistently require fewer lines of code than their corresponding generated C source code across evaluated modules, demonstrating improved developer productivity.
- **Performance Optimizations:** The artifact validates that specific optimizations produce measurable performance improvements. The evaluation results allow for a qualitative comparison against the trends reported in the paper.

Contents

The artifact is organized as follows:

- `sysspec/`: Contains the high-level filesystem specifications and the logic for generating the filesystem implementation.
- `eval/`: Includes evaluation artifacts for comparing baselines against optimized versions.
- `data/` & `tests/`: Datasets and scripts used by evaluation workloads and validation tests.
- `tools/` & `plot/`: Utility scripts for the pipeline and scripts for generating the figures.
- `gen.py`: The main script to run the generation pipeline and functional validation.
- `eval.py`: The main script to execute benchmarks and reproduce evaluation results.

Hosting

The artifact is hosted on GitHub (branch `main`): https://github.com/LLMNativeOS/specfs-ae. The repository includes a detailed `README.md` file explaining the specific claims and expected results.

Requirements

- **Operating System:** Linux is required (tested on Debian/Ubuntu).
- **Hardware/Software:** The system must support FUSE (Filesystem in Userspace). Required packages include `fuse`, `libfuse-dev`, `gcc`, `make`, and `python3`.
- **Python Environment:** The project uses `uv` for dependency management.
- **API Access:** An API key is required for the LLM backend. The artifact supports Google AI (Gemini, recommended) or DeepSeek.

B Case Study: `dentry_lookup`

We take `dentry_lookup` function as an representative example to illustrate the specification and the code generated from it.

B.1 Specification

Phase 1: Initial Implementation. Provide a complete C file that implements the `dentry_lookup` operation. You can use information from [RELY], [GUARANTEE], and [SPECIFICATION] as described below. Please output only the resulting file.

[RELY]

Predefined Structures/Functions

```
struct qstr {
    unsigned int hash;
    unsigned int len;
    const unsigned char *name;
};
struct dentry {
    struct qstr d_name;
    struct dentry *d_parent;
    struct hlist_node d_hash;
    atomic_t d_count;
    spinlock_t d_lock;
};
struct hlist_head { /* ... */ };
struct hlist_node { /* ... */ };

struct hlist_head* d_hash(struct dentry* parent,
↪  unsigned int hash);
int memcmp(const void *s1, const void *s2,
↪  size_t n);
int d_unhashed(struct dentry* dentry);

#define hlist_entry(ptr, type, member)
↪  container_of(ptr,type,member)
```

[GUARANTEE]

API Compliance: The function must have the exact signature declared below:

```
struct dentry * dentry_lookup(struct dentry *
↪ parent, struct qstr * name);
```

[SPECIFICATION]
Precondition: `parent` and `name` are valid pointers.
Postcondition: The function's behavior depends on whether a matching dentry is found.

Case 1 (Success) If a child dentry of `parent` is found with a name that matches `name`, and this dentry is currently active (not unhashed), then:

- The reference count (`d_count`) of the found dentry is incremented.
- A pointer to the found dentry is returned.

Case 2 (Failure) If no active child dentry of `parent` with a matching name is found, the function returns `NULL`.

System Algorithm:

1. Extract the hash, length, and string from the `name` parameter.
2. Use the `d_hash` utility to find the correct hash bucket (`hlist_head`) associated with the `parent` dentry.
3. Iterate through each dentry in the hash bucket in a loop.
4. For each dentry, perform the following checks:
   a. First, compare the hash value with `name->hash`. If they don't match, skip to the next dentry.
   b. Next, check if `dentry->d_parent` is the same as the input `parent`. If not, skip.
   c. Perform a full name comparison: compare the lengths (`dentry->d_name.len` and `name->len`) and then use `memcmp` to compare the string content. If the names do not match, skip to the next dentry.
   d. If all checks pass, verify that the dentry is not unhashed using `d_unhashed()`.
   e. If it is not unhashed, this is a successful match. Break the loop.
5. If a match was found, increment its `d_count` and return it. Otherwise, return `NULL`.

Phase 2: Concurrency Refinement. Please refine the above `dentry_lookup` function to correctly handle locks for concurrency. Please output only the resulting code. You can rely on the following information.
[RELY]

Predefined Structures/Functions

```
// Enters an RCU read-side critical section
void rcu_read_lock(void);
// Exits an RCU read-side critical section
void rcu_read_unlock(void);
// Safely dereference a pointer in an RCU
↪ critical section
struct hlist_node* rcu_dereference(struct
↪ hlist_node* p);
// Acquires a spinlock
void spin_lock(spinlock_t *lock);
// Releases a spinlock
void spin_unlock(spinlock_t *lock);
// Increment a counter atomicly
void atomic_inc(atomic_t *v);
```

The locking algorithm for `dentry_lookup` has three main components:
Component 1: RCU-Protected Traversal
Precondition: No RCU lock is held.
Postcondition: No RCU lock is held.
System Algorithm:

1. Before starting the search, call `rcu_read_lock()` to enter an RCU critical section.
2. After the search is complete (whether a dentry is found or not), call `rcu_read_unlock()` to exit the critical section.
3. Pointers within the hash list must be accessed via `rcu_dereference()` to ensure the most up-to-date value is read safely.

Component 2: Per-Dentry Spinlock
Precondition: The code is inside an RCU read-side critical section.
Postcondition: In all code paths after the lock has been acquired (e.g., after a successful match, after a failed re-check, or after a failed name comparison), the `dentry->d_lock` is released.
System Algorithm:

1. Inside the hash list traversal loop, after finding a candidate dentry (i.e., its hash matches), acquire its individual spinlock.
2. **Critical Re-check:** Immediately after acquiring the lock, re-check that `dentry->d_parent` still points to the `parent` you are searching under. If the parent no longer matches, release the lock and continue to the next dentry.
3. While holding the spinlock, perform the full, definitive name comparison. The lock ensures that `dentry->d_name` will not change.
4. If the name matches and the dentry is not unhashed, increment its reference count. This action must be performed before releasing the lock.

Component 3: Atomic counter Incrementation
Precondition: No lock is held on the counter.
Postcondition: The counter of the dentry is atomicly incremented. No lock is held on the counter.

B.2 Code Snippet

Phase 1: Initial Implementation.

```
struct dentry * dentry_lookup(struct dentry *
↪  parent, struct qstr * name) {
    unsigned int hash = name->hash;
    unsigned int len = name->len;
    const unsigned char *str = name->name;

    struct hlist_head *head = d_hash(parent, hash);
    struct hlist_node *node;
    struct dentry *found = NULL;

    for (node = head->first; node != NULL; node =
    ↪  node->next) {
        struct dentry *dentry = hlist_entry(node,
        ↪  struct dentry, d_hash);

        if (dentry->d_name.hash != hash)
            continue;

        if (dentry->d_parent != parent)
            continue;

        if (dentry->d_name.len != len)
            continue;

        if (memcmp(dentry->d_name.name, str, len) !=
        ↪  0)
            continue;

        if (d_unhashed(dentry))
            continue;

        found = dentry;
        break;
    }

    if (found) {
        found->d_count.counter++;
        return found;
    }

    return NULL;
}
```

Phase 2: Refined Implementation.

```
struct dentry * dentry_lookup(struct dentry *
↪  parent, struct qstr * name) {
    struct hlist_node *node;
    struct dentry *found = NULL;
    unsigned int hash = name->hash;

    rcu_read_lock();

    struct hlist_head *head = d_hash(parent, hash);

    for (node = rcu_dereference(head->first); node
    ↪  != NULL;
         node = rcu_dereference(node->next)) {

        struct dentry *dentry = hlist_entry(node,
        ↪  struct dentry, d_hash);

        if (dentry->d_name.hash != hash)
            continue;

        spin_lock(&dentry->d_lock);

        if (dentry->d_parent != parent) {
            spin_unlock(&dentry->d_lock);
            continue;
        }

        if (dentry->d_name.len != name->len
            memcmp(dentry->d_name.name, name->name,
            ↪  name->len) != 0) {
            spin_unlock(&dentry->d_lock);
            continue;
        }

        if (d_unhashed(dentry)) {
            spin_unlock(&dentry->d_lock);
            continue;
        }

        atomic_inc(&dentry->d_count);
        spin_unlock(&dentry->d_lock);

        found = dentry;
        break;
    }

    rcu_read_unlock();

    return found;
}
```

C DAG-Structured Patches for Features

Fig. 14 presents the DAG structures of patches used to implement the ten new features of SpecFS.

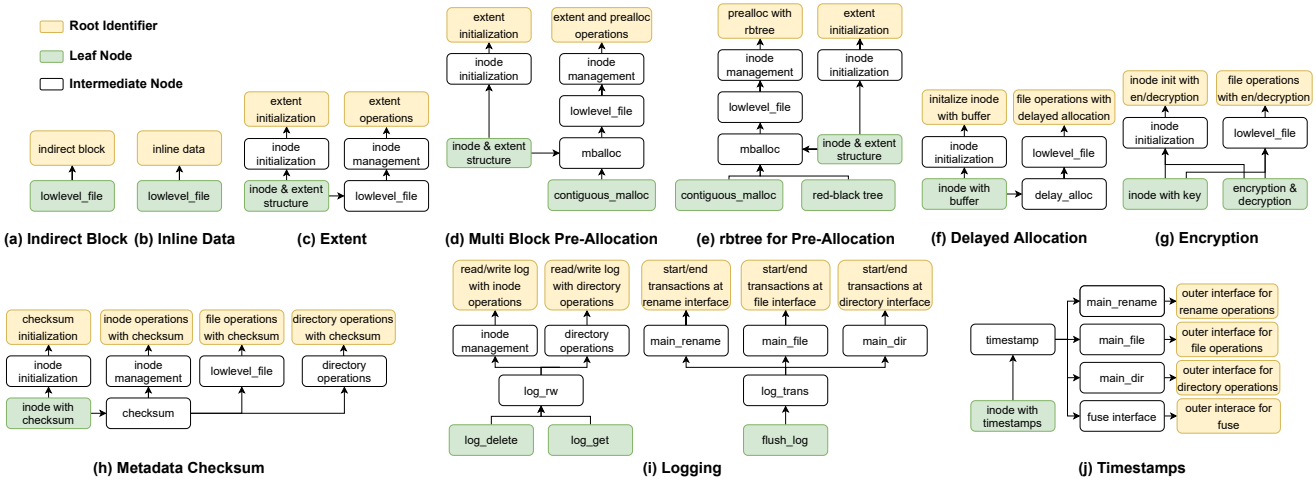


Figure 14: **Features implemented by SpecFS.** *For simplicity, a node in the figure may encompass multiple modules of specifications. "Root Identifier" indicates that this node is the root node, along with its associated logic.*