Fork, Explore, Commit: Linux Primitives for AI Agents Exploration

Cong Wang[1], Yusheng Zheng[2]

[1]Multikernel Technologies, Inc. · [2]eunomia-bpf / UCSC

Open Source Summit 2026

Agenda

1. **Background**: AI agents needs support for fork and exploration
2. **Problem**: execution leaves real filesystem and process side effects
3. **Requirements**: what fork-explore-commit needs from the OS
4. **Design**: branch contexts as the abstraction
5. **Implementation**: BranchFS in userspace, `branch()` in the kernel
6. **Status**: latency, demo, limitations, roadmap

What Is an AI Agent?

An **AI agent** is a control loop:

1. **Reason** with an LLM
2. **Act** in a local workspace with tool calls (shell commands, file edits)
3. **Observe** the result and repeat

Examples

Tool class	What it does locally
Claude Code, SWE-agent, OpenHands	edit your repo and run your tests
Aider, Cursor agents	same loop inside an editor
Devin, OpenAI Codex CLI	same loop on a hosted machine

Agent Exploration and Forking

Agents increasingly try **multiple paths** to solve a problem:

Pattern	What it does
Parallel Work	Try several solutions at once, keep the best result
Tree-of-Thoughts	Explore a tree of reasoning paths, prune weak ones
RL rollout	Run trials and score the outcome as reward
Speculate	Start likely paths early

Example: agent tries 3 candidate bugfixes on the same repo; commit only the one whose tests pass.

Agents on Linux: Processes with Effects

From `strace`, an agent looks like a developer typing fast:

```
execve("/bin/sh", ["sh", "-c", "pytest -x"], ...)
openat(AT_FDCWD, "src/parser.py", O_WRONLY|O_TRUNC)
write(3, "def parse(s):\n    ...", 4096)
execve("/usr/bin/git", ["git", "apply", "fix.patch"])
execve("/usr/bin/npm", ["npm", "install", "lodash"])
unlink("node_modules/.package-lock.json")
```

What this means for Linux

- They run as **ordinary processes**
- They produce **unpredictable side effects**: dirty trees, installed packages, build artifacts, modified dotfiles

The OS sees a normal Unix workload. The agent's *intent* ("this is one of three things I'm trying") is invisible.

Runtime Requirements for Agentic Exploration

#	Requirement	Why
R1	**Isolated parallel execution**	Concurrent paths modify same files
R2	**Hierarchical nesting**	Tree-of-Thoughts explores sub-variants
R3	**Complete filesystem coverage**	Capture *all* modifications, not just tracked files
R4	**Lightweight, unprivileged, portable**	Sub-ms creation, no root, any FS (ext4, XFS, NFS...)
R5	**Coordination**	For multi-agent, reliable termination, sibling isolation

No existing Linux mechanism satisfies these requirements. Let's walk through why.

Existing Solutions Fall Short

Hack	Why it hurts
`cp -r workspace branch1/`	10 GB monorepo -> slow, disk hog
`git stash` per attempt	Misses `node_modules`, build artifacts
Docker container per try	Heavyweight, needs daemon, root
One workspace, retry serially	No parallelism, wall-clock loss
`chroot` + bind mounts	Privileged, racy setup, no atomic commit

What we actually need:

- **One namespace** the agent lives in
- **N isolated branches** of it
- **Nested branches** for recursive exploration
- **No root**, **portable**

Let's see what Linux gives us today, and why it is not enough.

Filesystem Branching in Linux Today

OverlayFS

```
sudo mount -t overlay overlay \
  -o lowerdir=/repo,upperdir=…,workdir=… \
  /mnt/branch1
```

✓ Per-branch view · ✓ All modifications captured · ✓ Portable

✗ `sudo mount` required (rootless overlay is fragile)

✗ **Cannot commit changes back**: `rsync upperdir/ → lowerdir/` skips whiteout deletions

✗ **No automatic cleanup for losing branches**

✗ Nesting is complex and easy to break

Btrfs / ZFS subvolumes

```
$ btrfs subvolume create /repo
$ btrfs subvolume snapshot /repo /repo-b1
$ btrfs subvolume snapshot /repo /repo-b2
```

✓ O(1) creation · ✓ Block-level CoW · ✓ Nested subvolumes first-class

✗ **FS-locked**: your CI runs ext4, your colleague's laptop runs ext4

✗ **Cannot commit changes back**: `btrfs subvolume promote` doesn't exist

✗ **NFS / tmpfs / overlayfs** unsupported

✗ ZFS: `zfs promote` inverts parent ↔ child (wrong shape)

Both come close. Both miss **committing changes back**, **discarding losing branches**, and either **unprivileged** or **portable** operation.

Process Isolation in Linux Today

Each primitive does one thing well, none does the whole job:

Primitive	What it gives us	Cost
PID namespace	Reliable kill of all descendants	PID 1 init overhead
Mount namespace	Private view of `/mnt/workspace`	Needed anyway for FS isolation
cgroup v2	Reliable group termination via `cgroup.kill` (5.14+)	Setup, often needs root
`clone3()`	Compose namespaces at process creation	Multi-step, no atomicity
`setpgid` / `setsid`	Process groups	Escapable, child can `setsid()` and leave

The kernel exposes the right **ingredients**, but not one operation that **combines them safely**.

Composition is not straightforward

To make one branch in userspace

```
1. Mount the FS branch
2. unshare(CLONE_NEWNS)
3. clone3(NEWPID|NEWNS)
4. Move PID into cgroup
5. Install signal/ptrace fence
   (no primitive, DIY)
```

Each step can fail. Each gap between steps is a **race window**; making cleanup reliable adds overhead.

The bug between steps 3 and 4

```
parent: clone3() returns PID 12345
parent: ... about to add 12345 to cgroup
12345:  fork() → child 12346
parent: cgroup_add(12345), but 12346 is loose
parent: later, cgroup.kill, 12346 survives
parent: /proc-walk to find it. Maybe.
```

The forked grandchild escapes the cgroup. Reliable cleanup now needs a PID-1 babysitter or a single atomic primitive.

The Atomic Composition Problem

What we keep finding

Every existing OS primitive does **one thing well**: but agentic exploration needs **all of them, composed atomically**:

```
┌── FS branch (per-branch upperdir)
│
├── new mount namespace
│
├── new PID namespace (or cgroup)
│
├── signal/ptrace fence vs. siblings
│
└── child PID returned to parent
```

Each step in userspace = a **race window** + a **partial-failure path**.

Precedent: why `clone()` exists

- Before `clone()`, threads were almost buildable from `fork()` + shared memory + signals.
- Linux added `clone()` because that composition needed to be **atomic**.
- Branch contexts make the same argument for **filesystem state + process isolation**.

> The missing piece is not another sandbox. It is an OS-level mechanism: **branch()**. The pieces exist, but Linux lacks one operation that combines them safely.

Branch Contexts: Our Solution

Definition

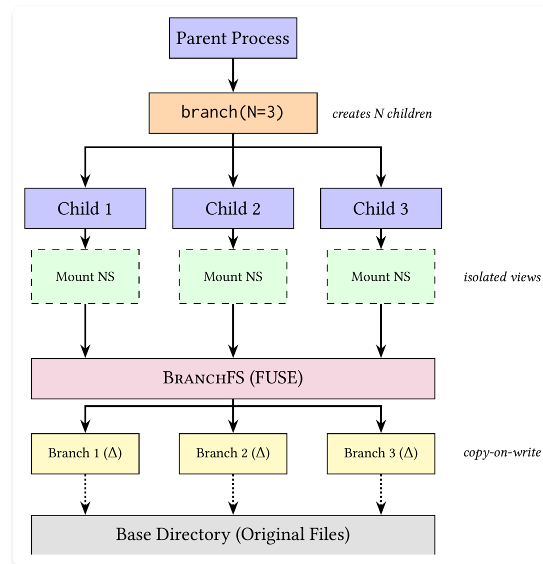
A **branch context** = CoW filesystem view (Δ_i) + confined process group

```
Fork ──▶ Explore ──▶ Commit (winner)
                 └──▶ Abort  (losers)
```

Four core properties

1. **Frozen origin**: parent read-only while branches exist
2. **Parallel isolated execution**: N siblings, fully isolated
3. **First-commit-wins**: siblings auto-invalidated
4. **Nestable**: a branch may fork sub-branches

Architecture



branch() = process coordination (kernel); **BranchFS** = filesystem CoW (FUSE userspace)

BranchFS - Speculative Branching Filesystem

What it does

- **CoW branch** per `@path` over any directory
- First write copies the file into the branch's delta
- **Atomic commit-to-parent**
- **Zero-cost abort** (delete the delta)
- **Nested branches** along a chain back to base
- **No root**, runs on any POSIX FS

Using it

```
$ branchfs mount --base /repo /mnt/work
$ branchfs create feature-a /mnt/work
$ cd /mnt/work/@feature-a
$ vim src/parser.py && make test
$ branchfs commit /mnt/work
```

Every branch is also reachable at `/mnt/work/@<name>/`. Multiple agents share one mount via `@branch` paths.

~4,400 LoC Rust · FUSE 3 · MIT/Apache-2.0 · `github.com/multikernel/branchfs`

File-Level Copy-on-Write

- First write to a file → **copy the whole file** into the branch's delta directory
- Subsequent reads and writes to that file are served from the delta
- Unmodified files: pass-through to the base (or an ancestor branch that modified it)

Trade-off vs. block-level CoW

	Block-level (Btrfs)	File-level (BranchFS)
First write of 1 MB file (1 byte changed)	~4 KB	1 MB
Implementation	Kernel-level COW	One FUSE handler
FS dependency	Btrfs-only	Anything

Why this works for agents

- Agent-touched files: source, config, small artifacts
- Typical size: KB to low MB
- Even huge `node_modules` only matters on the first write
- 1 MB copy ≈ 200 μs, still 1000× smaller than an LLM call

Branch Chain Resolution & Tombstones

Looking up a file

For `open("/mnt/work/@b3/src/main.py")`:

```
1. Check @b3's Δ           ← found? serve here
2. Walk ancestors: @b3 → @b1 → base
3. Hit base directory      ← serve here
4. Encounter tombstone?    ← return ENOENT
```

Branches form a chain back to base. Lookups stop at the first hit.

Tombstones handle deletions

Deletion on a branch writes a sentinel:

```
@b3/Δ/.tomb/src/old.py
```

Without tombstones, deleting a file on a branch would let the base copy "reappear" through the chain on next lookup.

Portable by construction: all BranchFS needs from the underlying FS is a writable directory. Branch create = `mkdir`. Branch destroy = `rm -rf`.

Commit: Apply Winning Changes Atomically

A commit applies a winning branch's delta in six steps:

```
1. Collect modified files + tombstones from Δ
2. Apply tombstones to target Δ        (deletes first)
3. Copy modified files into target Δ   (then creates)
4. Increment target epoch counter      (single atomic op)
5. SIGBUS on losing mmap'd regions     (their state is now stale)
6. Losing branch's next FUSE op returns -ESTALE
```

Order matters: deletes before creates. Otherwise a delete-then-recreate sequence in the branch could miscompose against an unchanged parent file with the same name.

Cost and Performance

Abort: near-zero cost

- Delete the branch delta; nothing else to do
- No base-copy cleanup needed
- Cost scales with changed files, not workspace size

This is BranchFS's **first-commit-wins primitive**. The `branch()` syscall (later) drives commit/abort through ioctls without changing the semantics.

Branch creation: O(1)

Base size	Latency
100 files	292 μs
1,000 files	317 μs
10,000 files	310 μs

Commit & Abort: O(delta)

Mod. size	Commit	Abort
1 KB	317 μs	315 μs
100 KB	514 μs	365 μs
1 MB	2.1 ms	890 μs

Performance: FUSE Read Throughput

Mode	Read
Native ext4	8.8 GB/s
FUSE (default)	1.7 GB/s
FUSE passthrough	**7.2 GB/s**

82% of native with FUSE 3 passthrough.

50 MB file, 64 KB blocks.

What is FUSE 3 passthrough?

- Added in mainline kernel **6.9**
- Daemon hands the kernel a lower-FD
- Subsequent reads bypass the daemon entirely
- Used by BranchFS for **all unmodified files**
- Modified files still go through the daemon
- **Opt-in** (`--passthrough`); needs `CAP_SYS_ADMIN`

The "FUSE is slow" reputation comes from the default mode's 19% number. Passthrough closes the gap to ~82% with no application changes.

Why a Syscall? Userspace Is Not Enough

What BranchFS gives us today

Requirement	Status
R1 isolated views	✓ via delta layers
R2 nesting	✓ via branch chain
R3 complete FS coverage	✓ via FUSE layer
R4 lightweight, unprivileged, portable	✓ userspace FUSE
R5 coordination	**✗, userspace cannot do this safely**

Four of five checked. Coordination needs the kernel.

What the Kernel Has to Do

Five capabilities `branch()` provides, only two are even *possible* from userspace:

Capability	Userspace?
Atomic composition of FS branch + mount NS + process group	Impossible, race windows
Memory branching (page-table CoW for `BR_MEMORY`)	Impossible, kernel only
Reliable process termination of all branch descendants	Cgroups only, needs root
Sibling signal/ptrace fence (siblings can't `kill -9` each other)	PID ns only, PID 1 overhead
Atomic mount-namespace setup with `open_tree` + `move_mount`	Possible but fragile

One syscall composes all five atomically, with kernel-side cleanup on partial failure.

The `branch()` Syscall: Interface

```
long branch(int op, union branch_attr *attr, size_t size);
```

Three operations

op	who calls it
`BR_CREATE`	parent: fork N children, each in its own branch
`BR_COMMIT`	child: apply this branch, terminate losing branches
`BR_ABORT`	child: discard this branch

`bpf(2)`-style multiplexed union → ABI-extensible.

Composable flags for `BR_CREATE`

Flag	Effect
`BR_FS`	Mount namespace + FS branch (required)
`BR_MEMORY`	Page-table CoW for memory
`BR_ISOLATE`	Signal/ptrace fence between siblings
`BR_CLOSE_FDS`	Close inherited FDs

The `branch()` Syscall: Usage

```c
int mnt = open("/mnt/work", O_PATH);
pid_t pids[3];
union branch_attr a = {
  .create = {
    .flags = BR_FS,
    .mount_fd = mnt,
    .n_branches = 3,
    .child_pids = (uintptr_t)pids,
  }
};
int idx = branch(BR_CREATE, &a, sizeof(a));
if (idx == 0) { // parent: wait for winner
  while (wait(NULL) > 0);
} else {
  // child: idx is 1, 2, or 3
  if (try_fix(idx)) {
    union branch_attr c = {.commit = {0}};
    int r = branch(BR_COMMIT, &c, sizeof(c));
    if (r == -ESTALE) _exit(1);  // lost the race
  } else {
    union branch_attr ab = {.abort = {0}};
    branch(BR_ABORT, &ab, sizeof(ab));
  }
```

Filesystem-Agnostic via Generic ioctls

The contract

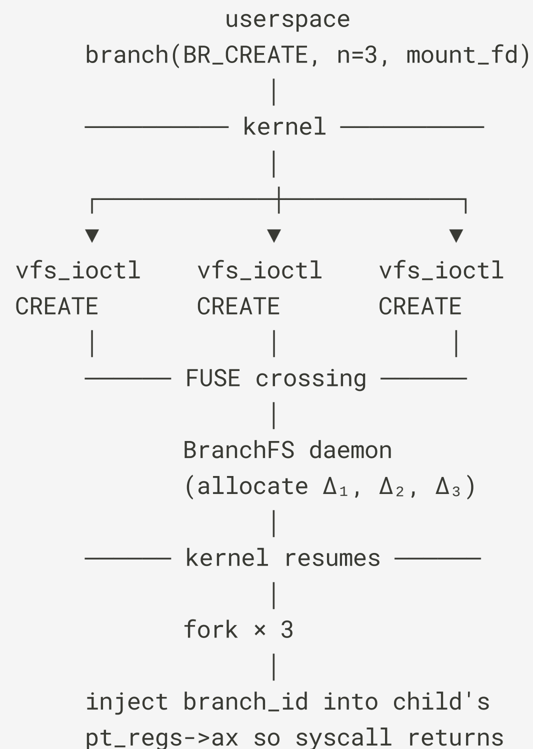
`branch()` does **no FS-specific work**. It delegates storage operations to the mounted branching filesystem:

```
FS_IOC_BRANCH_CREATE  _IO('b', 0)
FS_IOC_BRANCH_COMMIT  _IO('b', 1)
FS_IOC_BRANCH_ABORT   _IO('b', 2)
```

What this means

- **BranchFS** implements these ioctls today
- Other filesystems can implement the same contract later
- New backend = storage semantics, not a new syscall

Flow on `BR_CREATE`



Working Prototype: Vanilla Linux 6.17 + 3 Patches

The patch series

```
prototype/patches/
├── 0001-branch-add-UAPI-and-internal-headers.patch
├── 0002-branch-implement-syscall-and-ioctls.patch
└── 0003-branch-hook-copy_process-and-do_exit.patch
```

Three logical commits against `v6.17`.

Five-minute build

```
$ ./scripts/build-kernel.sh   # clone + patch + build
$ make -C test                # test program
$ ./scripts/build-rootfs.sh   # initramfs
$ ./scripts/run-qemu.sh       # boot + run

$ cargo build --release       # BranchFS
```

Capability status

Item	Status
`BR_CREATE` / `BR_COMMIT` / `BR_ABORT`	✅
CAS-based first-commit-wins	✅
Sibling SIGKILL	✅
`FS_IOC_BRANCH_*` ↔ BranchFS	✅
`BR_ISOLATE` fence	⚠️ flag accepted, fence not plumbed

Latency: Syscall Path Is Cheap

Measured in QEMU/KVM

61–70 μs

`BR_CREATE` : create branch state + fork child

12–25 μs

`BR_COMMIT` : CAS + filesystem commit path

~300 μs

BranchFS CLI branch creation in the paper

The big picture

- LLM/tool step: **100 ms – 10 s**
- `branch(BR_CREATE)` : **~70 μs**
- Ratio: **at least 1000:1**

These are sanity-check prototype numbers, not final benchmarks. The point is scale: branching is far below the latency of an agent step.

The Python Library: BranchContext

Patterns the library exposes

Pattern	Strategy
BestOfN	Run N, commit highest-scoring
Speculate	Race candidates, first success wins
TreeOfThoughts	Hierarchical nested branches

Each pattern manages branch creation, scoring, commit, and cleanup.

`github.com/multikernel/branching`

What an agent author writes

```
from branchcontext import BranchContext

ctx = BranchContext("/mnt/work")

# Best-of-N: try 3 fixes, commit best
result = ctx.best_of_n(
    n=3,
    task=lambda b: try_fix(b),
    score=lambda b: run_tests(b),
)
# Winner is already committed.
# Losers' deltas are gone.
```

Same Python API: BranchFS today, `branch()` later.

Demo: A Parallel Agent Run, Start to Finish

Transcript

```
$ branchfs mount --base $PWD /mnt/work
$ cd /mnt/work

# Race 3 fixes, first success wins
$ branching speculate \
    -c "./try_fix_a.sh && pytest" \
    -c "./try_fix_b.sh && pytest" \
    -c "./try_fix_c.sh && pytest"

[branching] @fix-b: success, committed
[branching] @fix-a, @fix-c: aborted
```

What happened

- Three candidate fixes ran in parallel
- Each branch wrote to its own delta
- `@fix-b` committed atomically
- Losing branches became stale
- Disk cost: changed files, not workspace copies

This is the whole fork-explore-commit loop in userspace today.

Status: What’s Shipping and What’s Not

Shipping today

- **BranchFS**: production-quality FUSE filesystem, Linux + macOS
- **BranchContext**: Python library, 7 patterns, on PyPI
- **branch() prototype**: 3 patches against v6.17, QEMU test harness, passing tests

Honest limitations

- **External side effects** (network, IPC) not rolled back on abort
- **Single-winner only**: no multi-branch merge
- **File-level CoW**: symlinks, hardlinks, FIFOs partial
- `BR_MEMORY` deferred, page-table CoW is real mm work
- **Nested branches** deferred in kernel prototype

Roadmap

Mainline RFC: port the `branch()` prototype forward and send the first patch series.

Sibling isolation: finish the signal / ptrace fence for hostile or buggy branches.

Nested branches: complete kernel support for recursive exploration.

External effect control: hold network / IPC until commit.

Beyond agents

With `n_branches=1`, `branch()` is also a generic **try-and-rollback** primitive:

- Package upgrades: try, abort if broken
- System config: try, revert if reboot fails
- Schema migrations: try, roll back on error

The fork-explore-commit lifecycle is more general than agents. Agents are just the loudest current use case.

How to Try It, How to Help

Try it

```
# BranchFS - works on any Linux today
$ cargo install branchfs
$ branchfs mount --base /repo /mnt/work
# Install branching tool
$ pip install BranchContext
```

(Optional) Kernel patches:

https://github.com/yunwei37/agentfs/tree/main/prototype/patches

Help wanted

- **Bugs / features** → GitHub issues
- **Kernel review** → LKML thread (soon)
- **New FS backends** → implement `FS_IOC_BRANCH_*`
- **Agent integrations** → BranchContext patterns

Resources

- BranchFS - `github.com/multikernel/branchfs`
- BranchContext - `github.com/multikernel/branching`
- Original Paper - https://arxiv.org/abs/2602.08199

Key Takeaways

1. **AI agents are ordinary Linux processes with extraordinary side effects.** The OS and Sandbox cannot tell when a process is one speculative path among many.

2. **Existing primitives don't compose atomically.** Filesystem branching, namespaces, cgroups, and signal fences need one kernel-level lifecycle.

3. **Branch context = CoW filesystem view + confined process group.** Fork, explore, commit; first winner lands, losers disappear.

4. **BranchFS works today; `branch()` is the kernel path.** Userspace prototype now, RFC direction next.

BranchFS: **github.com/multikernel/branchfs**

BranchContext: **github.com/multikernel/branching**

Thank You – Questions?

Cong Wang – cwang@multikernel.io

Yusheng Zheng – yzhen165@ucsc.edu

Open Source Summit 2026

BranchFS: github.com/multikernel/branchfs   BranchContext: github.com/multikernel/branching