The Parallel Developer

Three features in flight. One laptop. No context-switching tax.

Prakash Poudel Sharma · 2026

The hook

You are debugging a login bug. An agent is implementing avatar uploads. Another agent is extracting a billing service.

All on the same repo. All on the same laptop. None of them blocking the others.

Not a hypothetical. A daily working pattern.

What this talk is — and isn't

Is: a workflow design talk that happens to use AI.

Isn't: an AI hype talk.

Four tools do the heavy lifting:

- `git worktree` — branches as places
- **OpenSpec** — contracts before code
- **Beads** — a local-first task graph
- `.claude/rules/` — agent conventions in committed files

Why agentic coding (Part 1)

The shift: the human reviews, the agent executes.

- Copilot-style: the human types, the AI suggests.
- Agentic: the human hands off a task, the agent writes multiple files, the human reviews the PR.

The quality lever moves from suggestion accuracy to context quality.

The cost of one-thing-at-a-time

A normal day, before parallel:

- `git stash`, switch branch, run migrations, lose mental state.
- Wait on a code review — and do nothing useful while waiting.
- Pick up a half-finished branch tomorrow, re-orient for twenty minutes.

You aren't slow because you type slowly. You're slow because you serialise.

The thesis

Parallelism is unlocked by four things:

1. **Isolation** — every feature in its own directory, database, port.
2. **Contracts** — a spec the human approved before any code is written.
3. **A task graph** — one source of truth for what's next, what's blocked.
4. **Rules** — agent conventions checked into the repo.

The AI is the accelerant. The structure is the product.

Worktrees: branches as places (Part 2)

A git repo is `.git` + a working directory. But `.git` can support many working directories.

```
workspace/varicon/metapm/
  neo/                    ← main, never directly worked in
  neo-42-feat/            ← issue #42: avatar uploads
  neo-15-fix/             ← issue #15: login bug
  neo-33-refactor/        ← issue #33: billing extraction
```

Four directories. One `.git`. Switch context by switching terminal tabs.

Creating one

```
git worktree add ../neo-42-feat main -b 42-feat/add-avatars
cd ../neo-42-feat
bin/rails db:prepare
```

Naming is not cosmetic. `neo-<issue>-<type>/` encodes everything tooling needs:

- Port derived from issue number → `3042`.
- Database derived from branch type → `neo_development_wt_42-feat`.
- Agent context derived from directory name.

Isolation that matters

Per-worktree `.envrc.local` via direnv:

```
# .envrc.local in neo-42-feat/
export PORT=3042
export DATABASE_NAME=neo_development_wt_42-feat
export REDIS_DB=42
```

Three dev servers, three branches, three databases. Zero `git stash`. Zero "wait, which schema is loaded?"

What worktrees unlock

- **No stash dance** — switching contexts costs nothing.
- **Parallel CI locally** — three test suites at once.
- **Agents work in isolation** — they can't trample your debugging session.
- **Reviews stay rebuildable** — the branch is already checked out, already running.

Worktrees have been in git since 2015. They were waiting for this moment.

OpenSpec: contract before code (Part 3)

Freeform prompts produce freeform code.

> *"Add email change confirmation to the profile page."*

An hour later: 400 lines across 12 files. Logic in the controller. Cancel flow missing. Specs cover the happy path only.

The agent did its job. The spec was the problem.

Three commands

```
/opsx:explore    →  thinking partner, no files written
/opsx:propose    →  writes three spec files
/opsx:apply      →  implements from the spec
```

Explore sharpens the problem. Propose writes the contract. Apply executes against it.

The human reviews between propose and apply — before any code is written.

Anatomy of a proposal

```
openspec/changes/email-change-confirmation-ui/
  proposal.md    ← why + what changes + impact
  design.md      ← how + decisions + alternatives
  tasks.md       ← checklist, one box = one commit
```

Three files. Reviewed by a human. Then code.

Misunderstandings surface at minute five — not minute sixty.

Why specs first

Without a spec, review happens at PR time. The diff is 400 lines and you're rewriting cold.

With a spec, review happens at proposal time. The diff is three short markdown files and the context is hot.

Same review, shifted earlier. The cheaper round trip wins.

Beads: a local-first task graph (Part 4)

GitHub Issues is for cross-team communication. Great for that.

It is not great at: "what should I do next, right now, on this laptop, that isn't blocked?"

That is the question **Beads** answers. Single binary (`bd`). JSONL committed alongside your code. Agent-readable.

The killer command

```
bd ready
# bd-42.1  [P1] Add email_change_token column to users
# bd-15.3  [P2] Extract BillingService from UsersController
```

Only unblocked work. Sorted by priority. Two items, both actionable.

"What's next?" is now a single command — for you, and for the agent.

One bead, one worktree, one server

The pairing is the rule:

```
bd update bd-42.1 --status=in_progress
git worktree add ../neo-42-feat main -b 42-feat/add-avatars
cd ../neo-42-feat
bin/rails db:prepare
PORT=3042 bin/rails server
```

Claim → isolate → run. One bead at a time per worktree. The worktree name encodes the bead.

A tiny `bd` session

```
bd create --title="Add avatar uploads" --type=feat --priority=1
# created bd-42.1

bd dep add bd-42.2 bd-42.1   # 42.2 needs 42.1 first
bd ready                     # only 42.1 surfaces
bd update bd-42.1 --status=in_progress
# ... implement, commit, PR ...
bd close bd-42.1
bd export > .beads/issues.jsonl
git add .beads/ && git commit -m "Close bd-42.1"
```

The graph lives in git. Diffable. Reviewable. Rebuilt anywhere with `bd import`.

Agents that work (Part 5)

Compare two ways of starting a session.

Freeform: "Add avatar uploads to the profile page."

Structured: "Execute `bd-42.1` against `openspec/changes/add-avatar-uploads/`, in `neo-42-feat/`. Server on `:3042`. DB `neo_development_wt_42-feat`. Follow `.claude/rules/git-workflow.md`."

Same model. Different output. Every word in the second version was decided before the session started.

The ten-step loop

Step	Who	What
1	Human	Open GitHub issue
2	Agent	Read `CLAUDE.md` + `.claude/rules/*`
3	Agent	`/opsx:propose` → 3 spec files
4	**Human**	**Review and accept spec**
5	Agent	`/opsx:apply` → creates beads
6	Agent	`bd update --status=in_progress`
7	Agent	`git worktree add …`
8	Agent	Implement, commit per task, push, PR
9	Agent	`bd close`
10	**Human**	**Review and merge PR**

Two human steps. Both are review.

Rules in committed files

```
CLAUDE.md
.claude/rules/git-workflow.md
.claude/rules/task-management.md
.claude/rules/architecture.md
```

A snippet from `task-management.md`:

```
Before any implementation:
1. `bd ready` — pick highest priority unblocked.
2. `bd update <id> --status=in_progress`.
3. `git worktree add ../neo-<issue>-<type> main -b <branch>`.
4. Implement, commit, push, PR.
```

The agent reads it once per session. You don't repeat yourself.

What humans own, what agents own

Humans own:

- Spec review — does this proposal answer the right question?
- PR review — does this code reflect the spec?
- Taste, scope, architecture.

Agents own:

- Typing, scaffolding, plumbing.
- Following the rules they were given.
- Running tests, opening PRs, closing beads.

Neither side is doing the other's job.

Putting it together

```
issue  →  rules + spec  →  bead  →  worktree  →  PR
                ↑              ↑          ↑
            OpenSpec         Beads     git worktree
                          (with .claude/rules/* gluing it together)
```

A typical day:

```
:3042  neo-42-feat/       agent → avatar uploads (bd-42.1)
:3015  neo-15-fix/        you   → login edge case (bd-15.2)
:3033  neo-33-refactor/   agent → billing service (bd-33.1)
```

Try it tonight

Three concrete first steps — pick any one:

1. **Add one worktree** to a project you already work on. Run two dev servers side by side.
2. **Write one** `proposal.md` for the next thing you'd ask an agent to do. Read it before you prompt.
3. **Install** `bd`, create five beads from your current TODO list, run `bd ready`.

Each one stands alone. All three compound.

Read the series

The Parallel Developer, in five essays:

- Why Agentic Coding?
- Git Worktrees: Branches as Places
- OpenSpec: Contract Before the Code
- Beads: A Local-First Task Graph
- AI Agents That Work

Index: sharmaprakash.com.np/series/parallel-developer/

Thanks

Scan to read the series.

sharmaprakash.com.np


QR to the parallel-developer series