Parallel subagents: task graphs, coordination & token budgets

Key takeaways

1. Parallelism follows separability: launch only the independent ready set.
2. Claims, worktrees, receipts, and integration barriers replace shared state.
3. Keep control flow in code; multi-agent work can cost about 15× chat.

Common misunderstandings of parallel subagents

Which claim would you defend right now? Choose before the evidence.

"Different task names imply two subtasks are safe to run in parallel."

"Separate worktrees eliminate coordination conflicts between agents."

"Adding parallel workers speeds up any sufficiently large coding task."

Keep your choice. Look for the mechanism or counterexample that would make it fail.

In Week 4 the parallel workers were you

The vocabulary transfers to agent workers; the coordination cost does not disappear

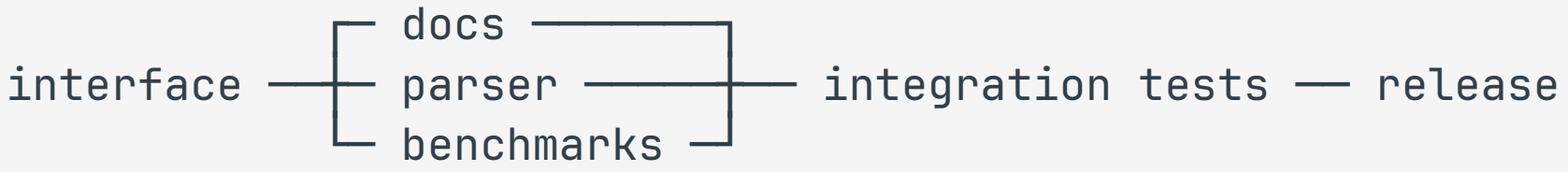
Parallel work (Week 4)	Parallel subagents (today)
You cycle attention across tmux sessions	A coordinator launches the whole ready set at once
A worktree isolates your task's files	A worktree isolates each agent's uncommitted edits
A receipt tells tomorrow-you how to resume	A receipt lets code accept or reject the work
A bad split costs your reorientation time	A bad split multiplies token cost and merge conflicts

The Week 4 test still decides: a task is ready when you can name its output and its merge point.

The task graph decides what may run now

Launch the ready set; everything else waits at a barrier

Ready set after the interface lands




```
ready set after interface = {docs, parser, benchmarks}
not ready                 = {integration tests, release}
```

Parallelism follows separability; it does not manufacture it.

Draw dependencies before you spawn

A task graph is an executable coordination contract, not documentation

needs/writes per task; ready = every need completed

```
tasks:
  interface:   {needs: [],          writes: [src/api.py]}
  parser:      {needs: [interface], writes: [src/parser.py]}
  docs:        {needs: [interface], writes: [docs/api.md]}
  integration: {needs: [parser, docs],
                writes: [tests/test_integration.py]}

ready = [t for t in tasks if set(t.needs) <= completed]
```

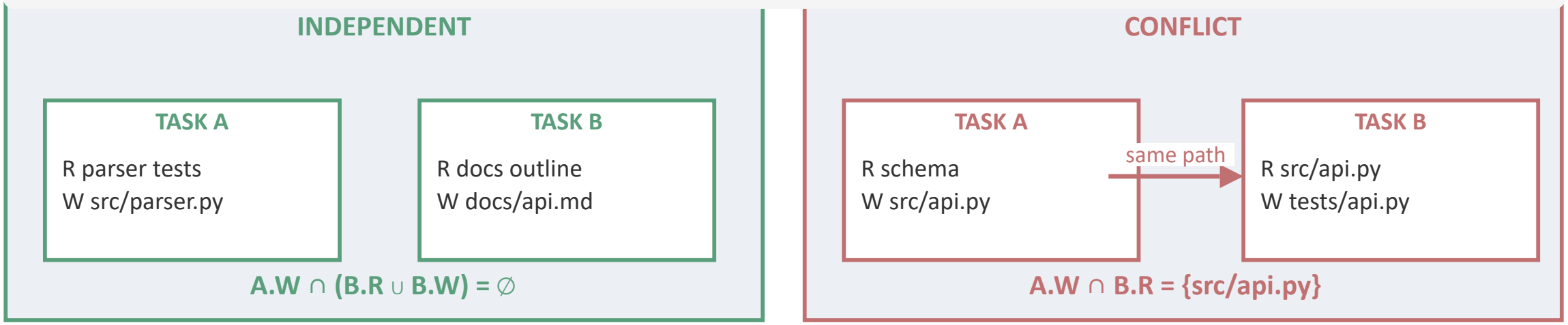
A coordinator may launch only the ready set. Everything else waits at a barrier instead of guessing against an interface still in motion.

Independence is a read/write-set question

Parallel-safe means more than different task names

The conflict test concurrent-index applies to threads, applied to tasks

```
conflict = (a.writes & (b.reads | b.writes)) |
           (b.writes & (a.reads | a.writes))
independent = not conflict
```



A shared evolving invariant couples agents even when their files differ.

Partition by evidence, not by wishful labels

Each worker needs a disjoint failure, component, corpus slice, or lens

Independent: separate evidence

```
# Agent A
run_tests("parser")
fix("src/parser.py")

# Agent B
audit_docs("docs/api.md")
fix("docs/api.md")

# Distinct inputs, outputs,
# and acceptance checks.
```

Coupled: one failing frontier

```
# Agent A ... Agent P
while not kernel_boots():
    fix_first_failure()

# Every worker sees the same bug,
# edits the same frontier, and
# invalidates everyone else's work.
```

A broad role such as “fix the compiler” is not a partition. Give each worker a disjoint failure, component, corpus slice, or verification lens.

Instapoll

Instapoll · Review · Oct 15 B

Which decomposition is most ready for parallel subagents?

- A. Independent components with frozen interfaces and separate acceptance tests
- B. Two tasks that both decide the same unresolved public data model
- C. One task whose input is the other task’s not-yet-designed output
- D. Several edits to one function without ownership boundaries

Compiler experiment: scale bought a real artifact

Sixteen Opus agents, two weeks, one 100K-line Rust C compiler

16	~2,000	2B / 140M	<$20K
parallel agents	**sessions**	**input / output**	**API cost**
Opus 4.6	over two weeks	tokens	capability experiment

Result: a roughly 100,000-line Rust C compiler that built Linux 6.9 on x86, ARM, and RISC-V. It was impressive, expensive, and still below expert Rust quality; generated code was less efficient than unoptimized GCC.

The experiment proves capability—not that sixteen agents are the default coding team.

source: anthropic.com/engineering/building-c-compiler

Sixteen agents can all be blocked by one dependency

Adding workers did not help until the harness changed the problem

Before: one giant kernel test

```
for worker in workers:
    worker.task = "make Linux boot"

# All workers hit the first
# compiler bug.
# Fixes overlap; merges collide;
# no worker owns a distinct failure.
```

After: an oracle creates slices

```
for worker, files in shards:
    worker.task = {
        "candidate": files,
        "oracle": "compile rest with GCC",
        "output": "failing subset to refine",
    }

# Different workers now see
# different failing evidence.
```

Anthropic reports that every worker initially hit the same kernel bug. Having 16 agents did not help until the harness changed the problem.

source: anthropic.com/engineering/building-c-compiler

Independent bugs partition cleanly; interactions do not

A trusted oracle converts one shared failure into many disjoint ones

Independent failure: test one file

```
def fails(claude_files):
    compile_claude(claude_files)
    compile_gcc(ALL - claude_files)
    return not boot_kernel()

for file in shard:
    if fails({file}):
        assign(worker, {file})
```

Interaction: reduce, then test pairs

```
# Start from a known failing set.
for chunk in partition(failing, n):
    if fails(chunk):
        return shrink(chunk)
    rest = failing - chunk
    if rest and fails(rest):
        return shrink(rest)

# Neither side may fail alone: a cause
# can cross the partition boundary.
for a, b in combinations(failing, 2):
    if fails({a, b}):
        return {a, b}
```

The GCC oracle exposes ordinary single-file failures. Subset and complement tests reduce a failing set; pair-aware delta debugging is still required when two files work independently but fail together.

source: anthropic.com/engineering/building-c-compiler

Anthropic’s experiment did not use worktrees

Any atomic claim plus any file isolation works; ours are mkdir and worktrees

Source design (simplified): containers + git

```
# One container per agent; bare repo mounted.
git clone /upstream /workspace
cd /workspace

echo "$AGENT" > current_tasks/$task.txt
git add current_tasks/$task.txt
git commit -m "claim $task"
git pull /upstream
git push upstream HEAD

work_on "$task"
git commit -am "finish $task"
git pull /upstream  # merge other agents
git rm current_tasks/$task.txt
git commit -m "release $task"
git push upstream HEAD

# A conflicting claim makes agent 2 retry.
```

Course adaptation: worktree + mkdir

```
# Local pattern; one shared claim root.
claim="$ROOT/claims/$task"
done="$ROOT/done/$task"

if mkdir "$claim" 2>/dev/null; then
  git worktree add "../wt-$task" HEAD
  run_agent "../wt-$task"
  mv "$claim" "$done"
else
  pick_another_task
fi

# mkdir is the local atomic claim;
# the worktree isolates index + files.
# Isolation does not fix task coupling.
```

source: experiment: anthropic.com/engineering/building-c-compiler; worktree/mkdir: course adaptation

Use code for known control flow; models for judgment

The compiler harness was shell, git, files, and tests — no model scheduled steps

Deterministic coordinator around nondeterministic workers

```
while pending:
    ready = task_graph.ready(completed)
    receipts = run_agents_in_parallel(ready)
    require_all(receipt.tests_passed for receipt in receipts)

    for receipt in receipts:
        merge(receipt.branch)

    run("pytest -q")          # integration barrier
    completed |= set(ready)

# Models solve tasks; code owns readiness, budgets, and barriers.
```

Compiler harness: shell + git + files + containers + tests; no model scheduled every step.

source: anthropic.com/engineering/building-c-compiler

Choose the smallest topology that fits

Spend an LLM coordinator only where decomposition or synthesis needs judgment

Topology	Use when	Coordinator
Sequential chain	Later work depends on earlier artifacts	Human or fixed script
Lead → workers	Decomposition changes as evidence arrives	LLM lead + coded limits
Shared task queue	Many ready tasks have strong tests	Code/files/git
Debate / critics	One high-value judgment needs adversarial views	Human synthesizes

Keep policy in code; spend an LLM coordinator only where decomposition or synthesis needs judgment.

Parallel reasoning is purchased with tokens

Anthropic’s production research system: ~15× chat, +90.2% on breadth-first tasks

+90.2%

multi-agent vs. single-agent on Anthropic’s internal breadth-first research eval

Anthropic production research data; relative token use

Pay 15× only when independent branches, latency, or task value can repay it.

source: anthropic.com/engineering/multi-agent-research-system

You own separability and integration

Your Responsibility	Agent's Responsibility
Define the task graph, shared invariants, and integration policy	Claim one ready task and stay inside its read/write boundary
Set concurrency and token budgets before launching workers	Commit an isolated artifact with a test receipt
Own the final integrated behavior—not just passing worker tests	Report dependencies or collisions instead of silently widening scope

Shared: The integration barrier is the truth: every merged batch reruns the system-level checks

Synthesis — parallelism follows separability

1. Start with a dependency graph and read/write sets; launch only tasks whose evidence and outputs are independent.
2. Use worktrees, claims, receipts, and integration barriers to coordinate shared code without shared mutable workspaces.
3. Keep deterministic control flow in code and budget model coordination explicitly—multi-agent work can cost about 15× chat.