Spec Kit: Specification-driven development

Make the work clear before asking an agent to build it

Session M5-02 | 3 hours 30 minutes

Agenda

Segment	Time
Why specification-driven development	8 min
Spec Kit project and artifact model	10 min
Requirements and clarification	12 min
Planning, checklists, tasks, and analysis	12 min
Implementation and convergence	10 min
Installation, demonstration, and lab handoff	8 min

A feature request is a starting point

> Users need to remove completed tasks from their normal working list without deleting task history.

This sounds clear until implementation starts:

- Does "remove" mean hide, archive, or delete?
- What happens when the same task is archived twice?
- Can callers still retrieve archived tasks?
- What result should an unknown task ID produce?

Code should not be where these decisions first appear.

Resolve decisions before implementation

Code-first path	Specification-driven path
The agent fills gaps while coding	The team resolves gaps before planning
Tests inherit hidden assumptions	Acceptance scenarios state expected behavior
Scope additions appear as code	Non-goals make additions visible
Review starts with implementation details	Review starts with requirements and test results

What Spec Kit provides

Layer	Purpose
CLI scaffolding	Initializes templates, scripts, memory, and an agent integration
Agent skills	Guide constitution, specification, planning, implementation, and review
Feature artifacts	Keep requirements, design decisions, tasks, and checks reviewable

The tool supplies a workflow. **The team still owns each product and technical decision.**

Initialize an existing project

```
specify init --here --force --integration copilot
```

Review the generated diff before using the workflow:

```
.specify/
├── memory/constitution.md
├── scripts/
└── templates/

.github/skills/
└── speckit-*/
```

`--force` allows initialization in a non-empty directory. It is not permission to accept every generated change without review.

Two artifact scopes

Project-wide	Feature-specific
`.specify/memory/constitution.md`	`specs/<feature>/spec.md`
Shared scripts and templates	`plan.md` and design files
Agent integration files	`checklists/` and `tasks.md`

The constitution applies across features. Each feature gets its own specification, design, tasks, and review trail.

The full workflow

```
Constitution
    ↓
Specify → Clarify
    ↓
Plan → Checklist → Tasks
    ↓
Analyze
    ↓
Implement ⇄ Converge
```

Clarify, checklist, and analyze are review gates. Use them when ambiguity or risk justifies the review.

Constitution: rules for how the project works

```
/speckit-constitution
```

A useful principle is testable:

Weak principle	Reviewable principle
Write high-quality code	Every accepted behavior has a focused automated test
Keep things simple	Use the Python standard library unless the plan records an approved exception
Avoid regressions	Existing create and list tests must remain in the suite

If a principle cannot affect a plan or review, delete it.

Specify behavior, not implementation

```
/speckit-specify
```

`spec.md` should answer:

- Who needs the behavior?
- What outcome do they need?
- Which scenarios test the behavior?
- What is outside scope?
- Which requirements are measurable?

Avoid class names, database choices, package names, and file paths at this stage.

Write scenarios a reviewer can challenge

```
Given an active task
When the caller archives its ID
Then the task keeps its ID and title
And the default list no longer returns it
```

```
Given an unknown task ID
When the caller archives it
Then the store returns a clear not-found result
```

Good scenarios expose decisions. "Archive works correctly" does not.

Clarify before choosing a design

```
/speckit-clarify
```

For the archive feature, clarification must settle:

Question	Accepted decision
Archive twice?	Return the same archived task
Unknown ID?	Raise `TaskNotFoundError`
Default list?	Exclude archived tasks
Historical access?	Support `include_archived=True`
Title during archive?	Preserve it unchanged

Record clarification decisions in the specification so future readers have one place to check the agreed behavior.

Plan: decide how the feature will work

```
/speckit-plan
```

The plan can now choose implementation details:

- Python 3.10 or later;
- standard library only;
- in-memory dictionary storage;
- an `archived` Boolean on each task;
- changes limited to `src/task_store.py` and its tests.

The specification owns behavior. The plan owns the technical approach.

Supporting design files have a purpose

Artifact	Question it answers
`research.md`	Which choices were made, and which alternatives were rejected?
`data-model.md`	Which entities, fields, rules, and state transitions exist?
`contracts/`	Which external interfaces must consumers follow?
`quickstart.md`	How can a reviewer check the planned behavior?

Not every feature needs every optional artifact. Keep the files that help someone implement or review the change.

A checklist reviews requirement quality

```
/speckit-checklist
```

A requirements checklist asks questions such as:

- Is repeated archive behavior defined?
- Is the default list behavior defined?
- Does the specification define an unknown-ID result?
- Are existing behaviors protected?

Checking an item means a reviewer found the requirement clear. It does **not** mean the code has been implemented or tested.

Tasks turn design into executable work

```
/speckit-tasks
```

Each task needs a place in the dependency order, a source requirement, and an exact file:

```
T001 [US1] Add archive state to new tasks in src/task_store.py
T002 [US1] Implement archive_task(task_id) in src/task_store.py
T003 [US1] Test default list filtering in tests/test_task_store.py
T004 [US2] Test repeated archive requests in tests/test_task_store.py
```

"Implement the feature" is not a task breakdown.

Analyze artifacts before implementation

```
/speckit-analyze
```

Analysis looks across `spec.md`, `plan.md`, and `tasks.md` for:

- a requirement with no implementation task;
- a task with no source requirement;
- a plan choice that contradicts accepted behavior;
- ambiguous or inconsistent terminology;
- missing coverage for a user story.

The command is read-only. Fix each issue in the artifact that owns the decision, then analyze again.

Implementation follows the accepted tasks

```
/speckit-implement
```

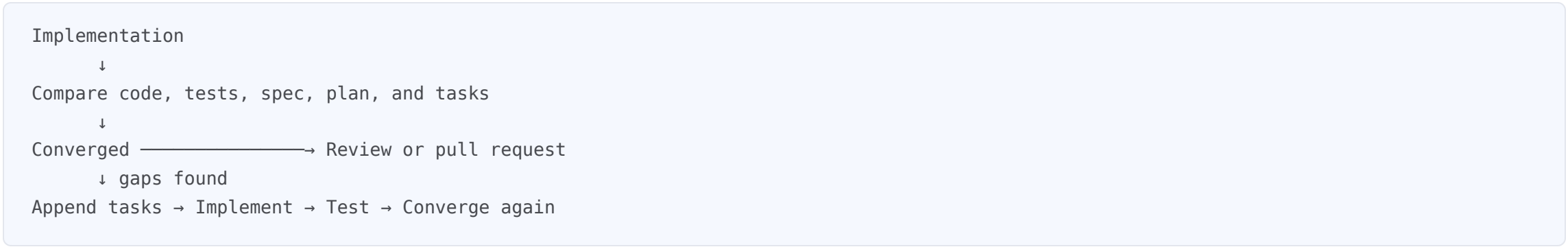
Before implementation:

1. Review and approve the requirements checklist.
2. Resolve material analysis findings.
3. Commit the accepted artifacts.
4. Check the implementation scope.

During implementation, connect tests and source code to the accepted user stories and requirements. The original prompt alone is not enough.

Compare implementation with accepted artifacts

```
/speckit-converge
```



Convergence does not rewrite requirements to match the code. It identifies the work needed to make the implementation and accepted artifacts agree.

`specify check` is not compliance validation

```
specify check
```

Use it to inspect the local Spec Kit environment and available tools.

It does not check whether the implementation meets the specification.
Use tests, review, `/speckit-analyze`, and `/speckit-converge` to compare the accepted artifacts and implementation.

Living specifications evolve accepted behavior

When requirements change, update the current feature artifacts in order:

```
spec.md
  ↓
plan.md
  ↓
tasks.md
  ↓
/speckit-analyze
  ↓
/speckit-implement
  ↓
/speckit-converge
```

Do not leave a restore requirement only in code or an issue if `spec.md` remains the contract.

Convert tasks into GitHub Issues

```
/speckit-taskstoissues
```

The generated issues should preserve:

- task IDs and user-story references;
- dependencies and parallel markers;
- exact file paths and expected work;
- the boundaries already accepted in `spec.md`.

Issues distribute the work. **They are not a second requirements source.**

Approved installation paths

Install only from an approved source and pinned release:

```
export SPECKIT_TAG='<approved-release-tag>'
uv tool install specify-cli --force \
  --from "git+https://github.com/github/spec-kit.git@${SPECKIT_TAG}"
specify version
```

Disconnected environments require the official air-gapped package prepared on a compatible connected machine. Stop when the approved installation or selected integration is unavailable.

Prepared demonstration

Show one deliberate gap:

1. Implement archive filtering.
2. Omit the `include_archived=True` test.
3. Run the existing unit tests.
4. Run `/speckit-converge`.
5. Inspect the task it appends.
6. Add the missing test and converge again.

This shows that passing tests and satisfying the specification are different claims.

Lab handoff

You will:

1. Run the starter tests.
2. Initialize Spec Kit in the Python project.
3. Specify and clarify task archiving.
4. Create the plan, checklist, and tasks.
5. Analyze before implementation.
6. Convert tasks into GitHub Issues.
7. Implement in stages and run the unit tests.
8. Converge until the artifacts and code agree.
9. Evolve the living specification to add restore behavior.

The lab ends with evolved artifacts, working code, passing tests, GitHub Issues, and a final convergence result.