Building on a Team

SECT.04-05.2026

- What’s hard about working on a team?
- How can we plan for & address the challenges?

There is no one-size-fits-all approach to building a team.

We'll discuss some general trends and lessons learned from my own experiences-- but this week in particular needs your engagement.

Please raise questions/thoughts from readings, prior team experience, and your own considerations for the project.

What is hard about building on a team?

- What are we building? With what tools?
- How do we manage decisions?
- Does the whole team have the same mental model? (theory building)
- How to we build at the same time/order dependencies?
- How do we manage complexity as things grow?
- What do we do when we disagree or can’t decide?

Challenges

- **Choices** - What do we build & with what tools?
- **Communication** - How do we stay on the same page while building?
- **Coordination** - How to we plan & track what needs to be built?
- **Complexity** - How do we keep things managable?
- **Culture** - What matters to us?
- **Continuity** - How do we ensure what we build can last?

Choices

- What choices need to be made?
- By whom?
- How do we make them?

Bike Shedding

aka "Law of Triviality"

"...just because you are capable of building a bikeshed does not mean you should stop others from building one just because you do not like the color they plan to paint it"

We spend more time on trivial details because everyone feels they can contribute to them.

https://thedecisionlab.com/biases/bikeshedding

https://bikeshed.org

Making Decisions

- **Decide what *needs* to be decided when.** Determine choices that need to be locked-in early vs. those that can remain in flux. Some choices can be left up to individual developers, or teams. Others are essential the entire team agrees upon.
- **Who decides?** Have a clear decision making process in place *before* decision is made to avoid bike-shedding.
- **Document alternatives & rationale!** This helps new (or existing) team members not suggest things that have already been debated & decided; and leaves a record for those that inherit what you built.
- **Don't overthink reversible decisions.** Some decisions are foundational and need to be decided, others might be an hour or two's work. Don't spend hours deciding something you can change in less time.

Must Agree

- Language & Libraries
- Version Control
- Code Quality Tools
- Deployment Environment

Can Differ

- Text Editor/IDE
- Development Environment (Docker, SSH, Local)
- Operating System

Hard to Change

- Language
- *Frameworks*

Can Delay

- Database
- License
- Architecture Details

Easier to Change

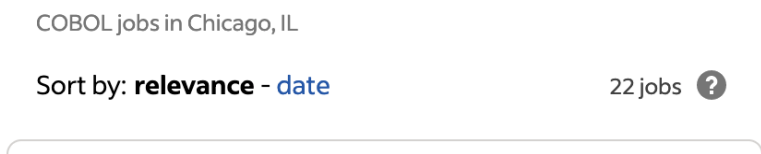
- Support Tools
- Libraries
- Specific Versions
- Cloud Platform*

*Assuming careful usage, beware accidental *lock-in*.

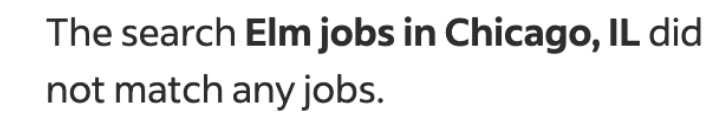
Balancing Familiar vs. New

"MySQL worked for us last time" vs. *"jQuery is so 2010s"*

COBOL (1960)



Elm (2012)



Use trusted technologies where appropriate.

Consider expected lifetime of project, team dynamics, and how hard it will be to change later.

Build vs. Buy vs. LLM

Buy

- Buy typically includes using free/open source software.
- 90s/00s: Distrust of OSS in many environments-- mostly overcome but still present in some environments.
- Perfect solution already exists (probably not!)
- No time to build or expertise on team.

Build

- Specific behavior essential to core goals of project.
- Introducing a dependency unnecessary/creates risk surface. (`left-pad`)
- Common, but bad reason: **NIH "Not-Invented-Here" Syndrome**: Noted developer aversion to "other people's code".
  - *"I don't trust it" / "It isn't perfect"*

LLM?

- Avoid a dependency for a *small* piece of code.

Communication & Coordination

How do we work independently while maintaining a consistent mental model & shared goal?

How do we understand what is to be built, and when?

(ideally with minimal meetings)

Reducing Friction

Plan upfront to align understanding of problem as much as possible. Meetings most valuable at start. If I had 20 hours of meetings for a 10 week project, I'd put 10 in the first two weeks.

Ensure team can maintain some visibility into what the rest of the team is building. Keep others' work in peripheral vision while your work is in focus.

Async Communication

- Issue Tracker
- Chat with threads (e.g. Slack)
  - **avoid private/1:1 chats!**
- Good documentation. (Documentation-driven development)
- Code Review / PR tools (e.g. GitHub)

The abandoned wiki/issue tracker/etc. is a sign that the team lacks processes encouraging them to use their tools. Re-evaluate processes & tools.

Processes

- **Scrum:** Sprint Planning, Daily Stand-ups, Review & Retrospectives
- **Kanban:** Needs discipline and/or to borrow from scrum.
- **Enhancement Proposals:** Common in FOSS, anyone can write technical proposal & team has process to evaluate & decide. / Kanban
- **Code Review:** Two sets of eyes. Every PR needs someone to sign off. Helps ensure consistent style & encourage interface discussions.

Ignored processes leave gaps, unused/neglected processes need to be **replaced**.

Coordination Tips

- Bring group together for important decisions like key interfaces & technology choices.
- Processes should be *easy to follow* and *provide value* to everyone involved.
- Ensure team communication happens in places everyone can access today & in the future.
- Better to make an overly-broad ticket than forget to record it at all. *"Someone needs to write the UI"*
- Self-govern: adjust processes in response to observed weaknesses.

Complexity

How do we keep complexity to a minimum, and manage the complexity that arises from thousands of lines of code and everything that comes with it?

Assume that no single person can be familiar with the entire codebase.

Where does complexity come from?

Typical mid-sized project:

- 50k lines of code
- 20-50 direct dependencies and another 100-1000+ indirect. (`create-react-app` introduced 1400 package dependencies at its worst.)
- 10-100 DB tables/models.
- Dozens of infrastructure dependencies.

Managing Complexity with Tools

To reduce decision fatigue and enforce uniform style, use linters & autoformatters. (`ruff` or `black`)

Reduce need to remember: configured to run automatically/on git commit.

Consider `ty` , `mypy` or `pyright` to add **typing** to your Python. Self-documenting interfaces:

```
>>> get_sales_records({"artist": "Nirvana", "title": "Nevermind", "year": 1991})
[["1991", "2026"], [100000, 1400000, 1500000, ...]]
```

vs.:

```
class Album(typing.NamedTuple):
  artist: str
  title: str
  release_year: int

def get_sales_records(albums: list[Album]) → dict[Album, dict[str, int]]:
```

Interfaces Encourage Compartmentalization

When we `import httpx` (or `requests`) we delegate understanding the nuances of HTTP to those authors.

```
import httpx

resp = https.get("https://example.com", headers={"User-agent": "test/123"})
if resp.status_code == 200:
    print(resp.text)
```

I am responsible for understanding the *interface*: `get` and `HttpResponse`.

Interfaces Enable Parallel Work

```
def load_data(filename: str) → Data: ...
def process_data(data: Data) → Data: ...
def visualize_data(data: Data) → VizImg: ...
```

If we agree on `Data` interface, all three can happen in parallel.

Avoiding Over-Engineering

What is the simplest interface that will cover my current needs?

Does this make it easy to do what I am trying to do?

Prioritize:

1. Correctness
2. Understandability
3. Maintainability
4. Performance

Complexity Traps

Premature Optimization

"*is the root of all evil*" - D. Knuth

Excessive Configurability

Avoids making decisions at expense of complexity.

"What if instead of picking library for X I left it configurable?"

YAGNI - You Ain’t Gonna Need It

Yak Shaving

Culture

How can we foster a team culture that allows for collaborative and individual success?

- Set clear expectations via roles & processes.
- Favor group communication.
- Make sure everyone’s contributions and time are treated as valuable.

Examples

- **"Decider" roles / code owners** - Delegate authority within codebase based on expertise. (Simplified graph.)
- **Code of Conduct** - Helpful to set some clear expectations around team behavior, especially for larger/open source teams.
- **Code Review** - ~~"This is awful"~~ - ~~"I would have written this by..."~~ - Catch small mistakes, acting as a team - Surface conversation about important choices

Purpose of Roles

Help ensure nothing slips through cracks, and to help streamline decision-making.

Do not need to be hierarchical, especially on smaller projects.

On small teams, usually quite fluid since 1:1 developer:role ratio isn't possible/desirable.

Recommendation for this year: assign fractional roles to each member.

To have functional specialization, usually need `developers ⩾ 2*number of roles`.

Without this, `bus_factor = 1`

Continuity

How can we ensure that the software remains usable and **maintainable**?

Continuity

- Software often lives far longer than anybody initially involved expected.
- Out-of-date documentation is often worse than no documentation.
  - Focus on processes (technical & human) that will keep documentation in sync.
  - Why things work the way they do is arguably more important than documenting how they work.
- You will inherit code where continuity wasn't a consideration, and wish you could start from scratch.
  - Rewrites are one of the biggest risks/source of failures.

Continuity Tools

Use **documentation generators** which extract code from docstrings to keep documentation up to date. Documentation & code that live side-by-side is much more likely to remain up to date.

Examples: `sphinx` & `zensical` (formerly `mkdocs`)

Similarly, tests that nobody runs are worth nothing. Ensure that your tests run on every git commit using **continuous integration** tools. It is also a good idea to ensure your tests have high **coverage**.

Examples: `GitHub actions` & `coverage.py`

Bus Factor: What if a team member wins the lottery/decides to hop on a bus & never return?

Bus factor is how many team members could hop on that bus before the project fails.

Avoid `bus_factor = 1`, redundant roles.

Documentation

At a minimum, a project should have a README that describes how to run it, and what it does. Even for yourself when you revisit it later.

Wiki vs. Inline Documentation: Who needs to contribute?

Module documentation vs. Architecture/Team Process documentation

Milestone 2

https://github.com/uchicago-capp-30320/sect-2026/blob/main/milestones/m2.md

Software Development Processes

Waterfall

Agile

Scrum / Kanban

Waterfall

"Ship out the instruction book before building the software"

Not a Recommendation!

The original paper by Royce (1970), aims to describe common practices-- advocates for more testing, earlier involvement of the customer, and other sensible modifications.

Agile Manifesto

https://agilemanifesto.org

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan

> That is, while there is value in the items on the right, we value the items on the left more.

12 Principles

https://agilemanifesto.org/principles.html

- Focused on continuous delivery.
- Responsive to change.
- Constant, sustainable pace.
- Design and simplicity.
- Self-organizing & governing teams.

Capital-A Agile

Scrum-masters, trainings, books, strict rituals.

small-a agile

"Chop things up small & deliver incrementally"

Mirrors UCD principles, continuous iteration, rapid feedback.

Scrum

Commonly 2 week sprints starting with sprint planning & ending with review & retrospective.

Heavy on ceremonies and roles. "Scrum Master" and mandatory meetings.

Expect to spend about one full day per sprint in meetings (first/last).

- Focus on *momentum* and metrics.
- Good for teams where tasks are not easily shared so order matters to avoid blockage.
- Good for projects with unclear goals that will come into focus as project evolves.

Kanban

Work winds up in loosely prioritized "swim lanes" (Backlog, TODO, In Progress, Blocked, Done, etc.)

Key Idea is to set limits on stages: e.g. max of `$team_size*2` items in in-progress state.

Allows team to self-govern.

- Focus on autonomy.
- Good for teams where tasks can be easily exchanged among team members.
- Good for projects with clear set of tasks ahead/ongoing maintenance.

Ritual: Point Estimation

Done collaboratively, should be to help distribute work.

Key: Split big issues into smaller ones.

Team "Velocity": points/sprint

Non-time measures, e.g. T-Shirt Sizes (S, M, L, XL), Fibonacci, Planning Poker

Iterate on Process

"At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."

–Agile Manifesto

Schedule time to talk about what is and isn't working and refine periodically.

Take what works, don't be overly-attached to any particular process.

Team Issue Debugging Tips

Workload Imbalances - Revisit estimation. Architecture.

Skill Differences - Pair program/give non-blocking tasks while someone learns.

Bottlenecks - Consider architecture / lightweight mocking.

Work Not Done - Increase check-ins (virtual). Revisit estimation.

Critical for this class: identify & raise to course staff early