CEN 5016: Software Engineering

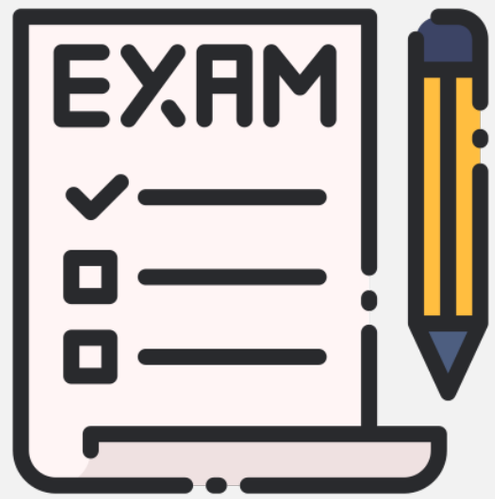
Fall 2026



University of Central Florida

Dr. Kevin Moran

Week 7: Midterm Exam Review



Midterm Exam Format

- 2 Parts, In-class exam, closed book, 200 points total
  - ***Part 1:*** Multiple Choice
    - 12-15 questions
    - Will test basic knowledge of concepts, select the best answer for each question
  - ***Part 2:*** Short Answer Questions
    - 4-5 questions
    - Concepts from class, SE scenarios, answer in a paragraph
- Covers all lectures through Agentic Software Engineering II (Oct 13)
- You will have the ***entire*** class period to complete the exam
- Please bring your UCF ID to the exam

Example Multiple Choice Questions

- Which of the following is NOT a tenant of Agile?
  - (a) Incremental Design/Development
  - (b) Inspect and Adapt Cycles
  - (c) Ignoring the Customer
  - (d) Collaborative workflows
- What is the name of the concept where someone looks for something where they think it will be?
  - (a) the spotlight effect
  - (b) the streetlight effect
  - (c) The candle effect
  - (d) the software effect

Example Short Answer Questions



- *Consider the following scenario: You are working on a development team that seems to have a lot of issues with reoccurring bugs in your codebase. Describe some concepts from class that might aid in this situation. Be sure to use at least two separate concepts.*

Week -1 Software Archeology & Anthropology



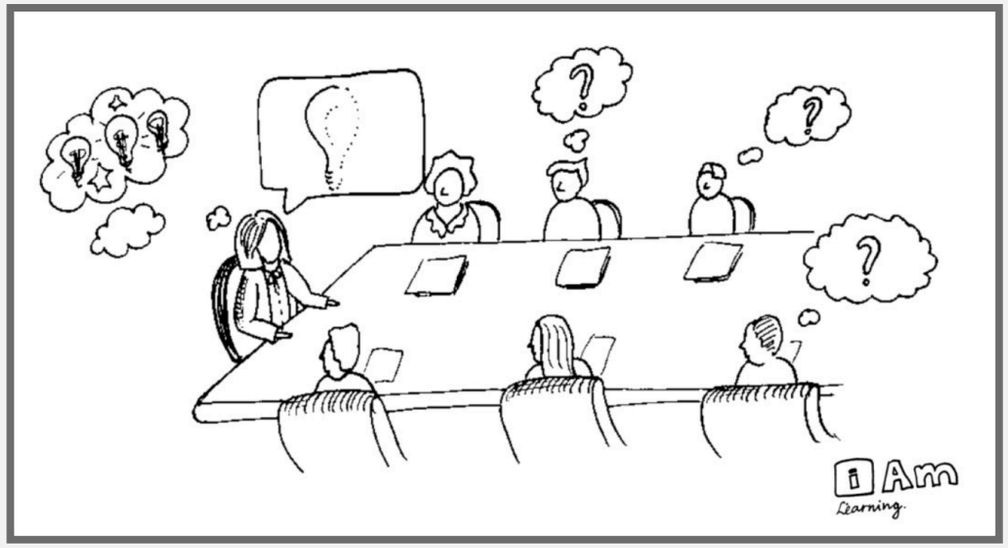
High-Level Strategies



- Leverage your previous experiences (languages, technologies, patterns)
- Consult Documentation, white papers
- Talk to experts, code owners
- Follow best practices to build a working model of a system

Why? Because of Tacit Knowledge

- *Tacit knowledge* or *implicit knowledge*—as opposed to formalized, codified or explicit knowledge—is knowledge that is difficult to express or extract; therefore it is more difficult to transfer to others by means of writing it down or verbalizing it.

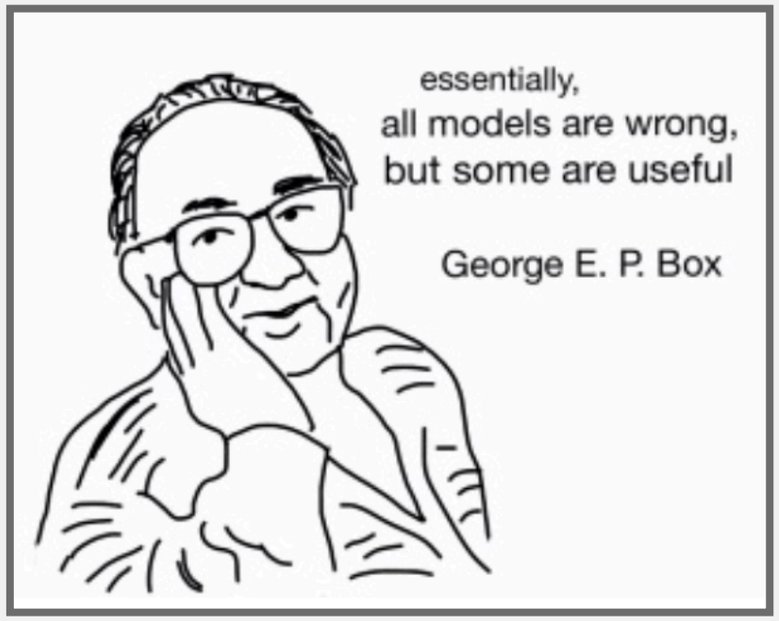


Today: How to tackle Codebases

- *Goal:* Develop and test a working model or set of working hypotheses about how (some part of) a system works

- *Working model:* an understanding of the pieces of the system (components), and the way they interact (connections)

- *Focus:* Observation, probes, and hypothesis testing
  - Helpful tools and techniques!



Steps to Understand a New Codebase



- Look at README.md
- Clone the repo.
- Build the codebase.
- Figure out how to make it run.
- What do you want to mess with?
  - Clone and own
- Traceability - Attach a debugger
  - View Source
  - Find the logs.
  - Search for constants (strings, colors, weird integers (#DEADBEEF))

Observation: Software is Full of Patterns

- File structure
- System architecture
- Code structure
- Names
- …



Check for updates

On the Naturalness of Software

Abram Hindle, Earl T. Barr, Zhendong Su
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
{ajhindle,barr,su}@cs.ucdavis.edu

Mark Gabel
Dept. of Computer Science
The University of Texas at Dallas
Richardson, TX 75080 USA
mark.gabel@utdallas.edu

Premkumar Devanbu
Dept. of Computer Science
University of California at Davis
Davis, CA 95616 USA
devanbu@cs.ucdavis.edu

***Abstract*—Natural languages like English are rich, complex, and powerful. The highly creative and graceful use of languages like English and Tamil, by masters like Shakespeare and Avvaiyar, can certainly delight and inspire. But in practice, given cognitive constraints and the exigencies of daily life, most human utterances are far simpler and much more repetitive and predictable. In fact, these utterances can be very usefully modeled using modern statistical methods. This fact has led to the phenomenal success of statistical approaches to speech recognition, natural language translation, question-answering, and text mining and comprehension.**

We begin with the conjecture that most software is also natural, in the sense that it is created by humans at work, with all the attendant constraints and limitations—and thus, like natural language, it is also likely to be repetitive and predictable. We then proceed to ask whether a) code can be usefully modeled by statistical language models and b) such models can be leveraged to support software engineers. Using the widely adopted n-gram model, we provide empirical evidence supportive of a positive answer to both these questions. We show that code is also very repetitive, and in fact even more so than natural languages. As an example use of the model, we have developed a simple code completion engine for Java that, despite its simplicity, already improves Eclipse's built-in completion capability. We conclude the paper by laying out a vision for future research in this area.

***Keywords*-language models; n-gram; natural language processing; code completion; code suggestion**

efforts in the 1960s. In the '70s and '80s, the field was reanimated with ideas from logic and formal semantics, which still proved too cumbersome to perform practical tasks at scale. Both these approaches essentially dealt with NLP from first principles—addressing *language*, in all its rich theoretical glory, rather than examining corpora of actual *utterances*, *i.e.*, what people actually write or say. In the 1980s, a fundamental shift to *corpus-based, statistically rigorous* methods occurred. The availability of large, on-line corpora of natural language text, including "aligned" text with translations in multiple languages,[1] along with the computational muscle (CPU speed, primary and secondary storage) to estimate robust statistical models over very large data sets has led to stunning progress and widely-available practical applications, such as statistical translation used by `translate.google.com`.[2] We argue that an essential fact underlying this modern, exciting phase of NLP is *natural language may be complex and admit a great wealth of expression, but what people write and say is largely regular and predictable*.

Our *central hypothesis* is that the same argument applies to software:

> *Programming languages, in theory, are complex,*

Observation: Software is Massively Redundant



- There is always something to copy/use as a starting point!



The Beginning: Entry Points



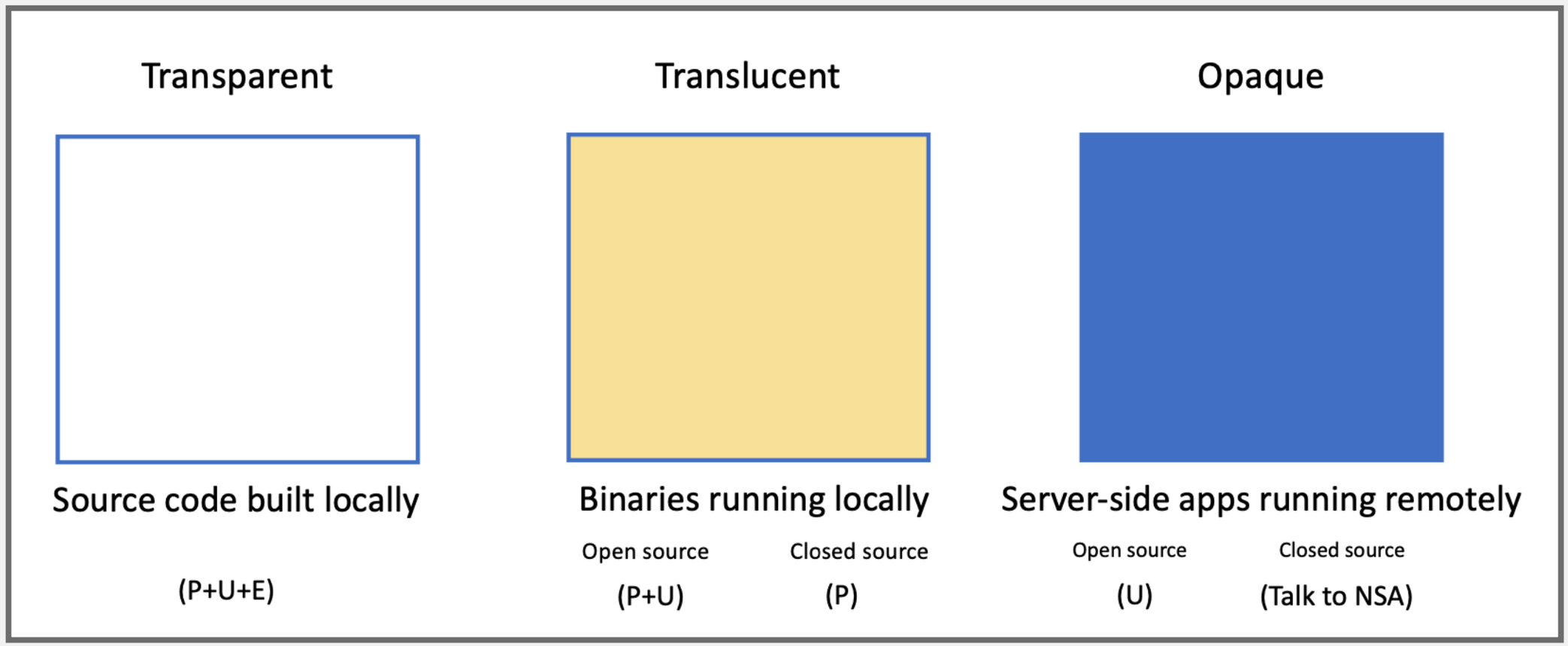
- Locally installed programs: run cmd, OS launch, I/O events, etc.
- Local applications in dev: build + run, test, deploy (e.g., docker)
- Web apps server-side: Browser sends HTTP request (GET/POST)
- Web apps client-side: Browser runs JavaScript, event handlers

Code Must Exist: But Where?



- Locally installed programs: run cmd, OS launch, I/O events, etc.
  - Binaries (machine code) on your computer
- Local applications in dev: build + run, test, deploy (e.g., docker)
  - Source code in repository (+ dependencies)
- Web apps server-side: Browser sends HTTP request (e.g., GET, POST)
  - Code runs remotely (you can only observe outputs)
- Web apps client-side: Browser runs JavaScript, event handlers
  - Source code is downloaded and run locally (see: browser dev tools!)

Can Running Code be Probed/Understood/Edited?



Information Gathering



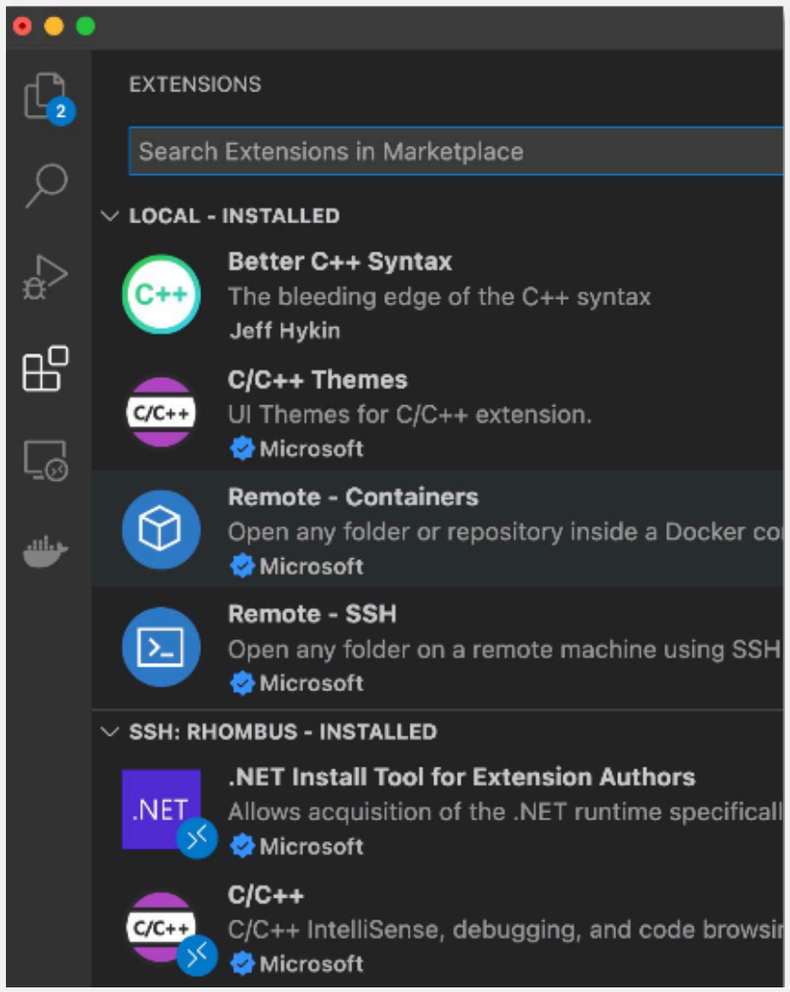
- Basic needs:
  - Code/file search and navigation
  - Code editing (probes)
  - Execution of code, tests
  - Observation of output (observation)

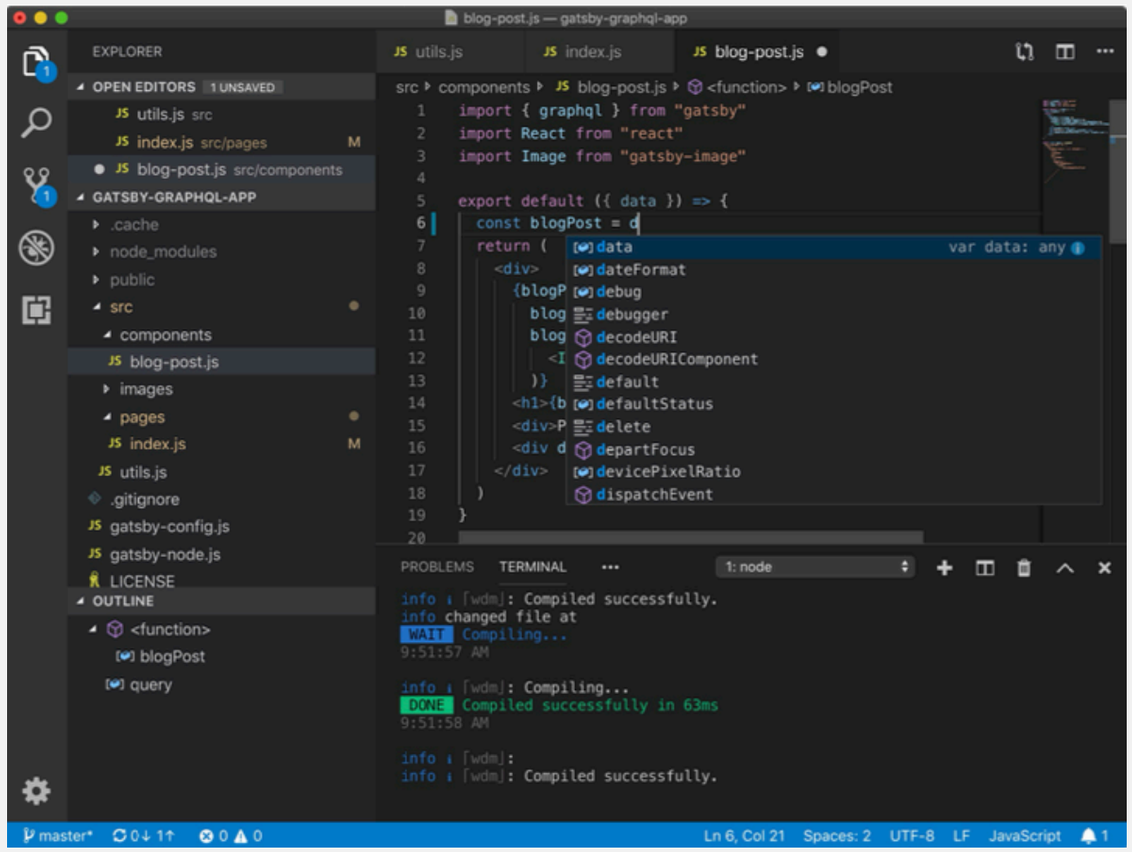
- At the command line: grep and find! (Google for tutorials)

- Many choices here on tools! Depends on circumstance.
  - grep/find/etc.
  - Knowing Unix tools is invaluable
  - A decent IDE
  - Debugger
  - Test frameworks + coverage reports
  - Google (or your favorite web search engine)
  - ChatGPT or LaMA

Static Information Gathering: Use an IDE!



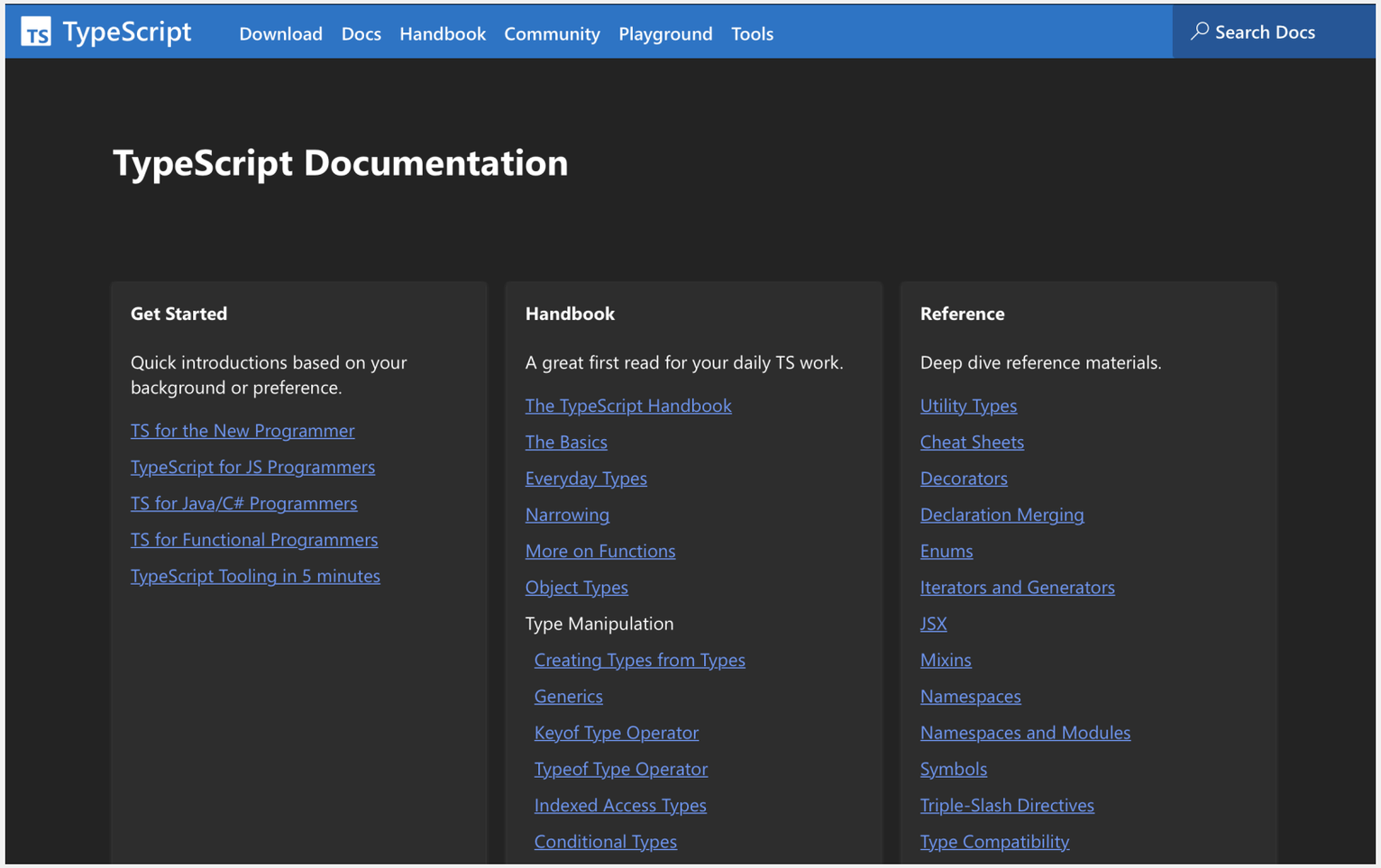




Consider Documentation and Tutorials Judiciously



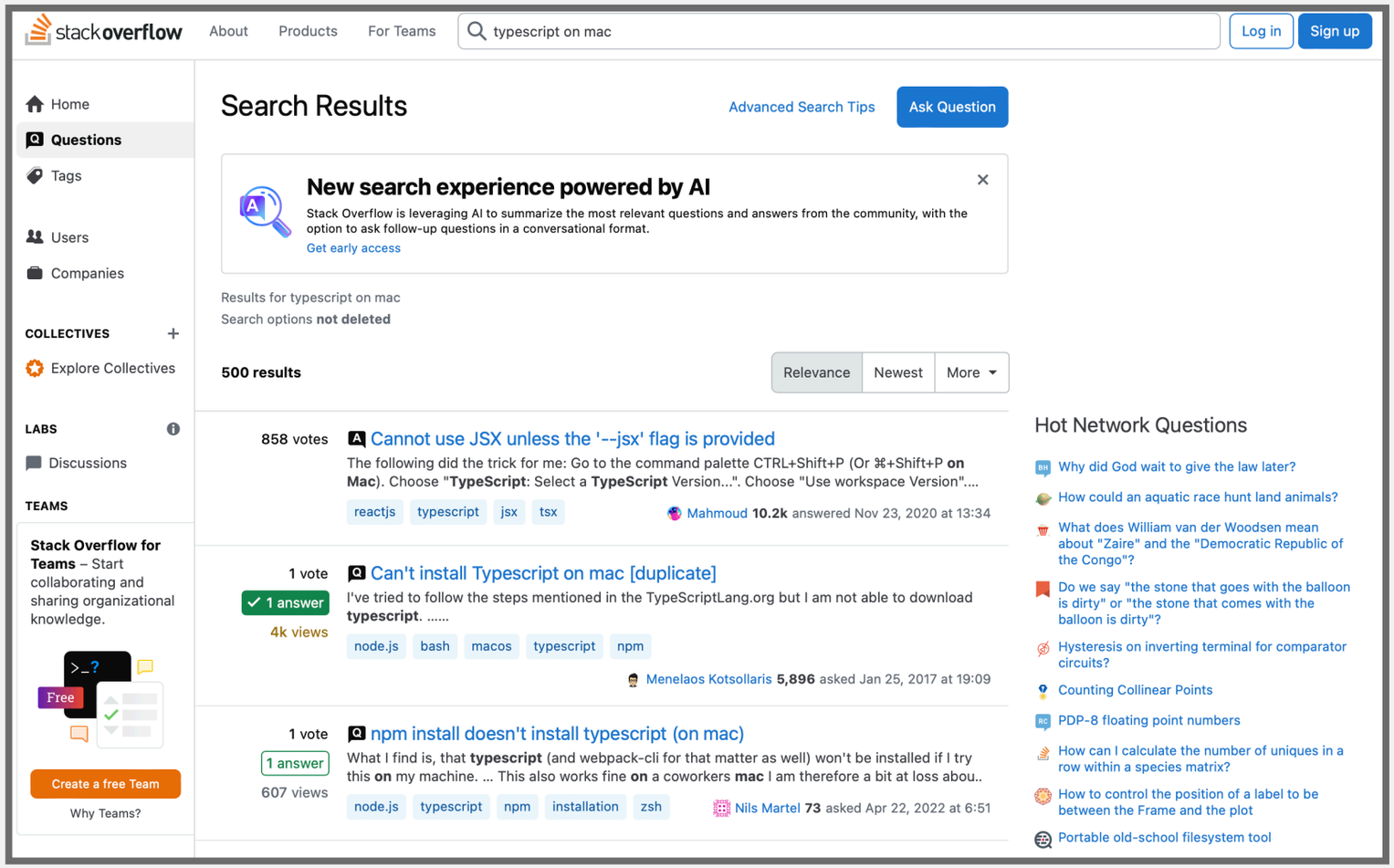
- Great for discovering entry points!
- Can teach you about general structure, architecture (more on this later in the semester)
- Often out of date.
- As you gain experience, you will recognize more of these, and you will immediately know something about how the program works
- Also: discussion boards; issue trackers



Discussion Boards and Issue Trackers



- Software is written by people.
- How can we talk to them?
- Fortunately, they probably aren't dead.
- So, you can report problems on GitHub.
- Or, ask them questions on StackOverflow.



Dynamic Information Gathering

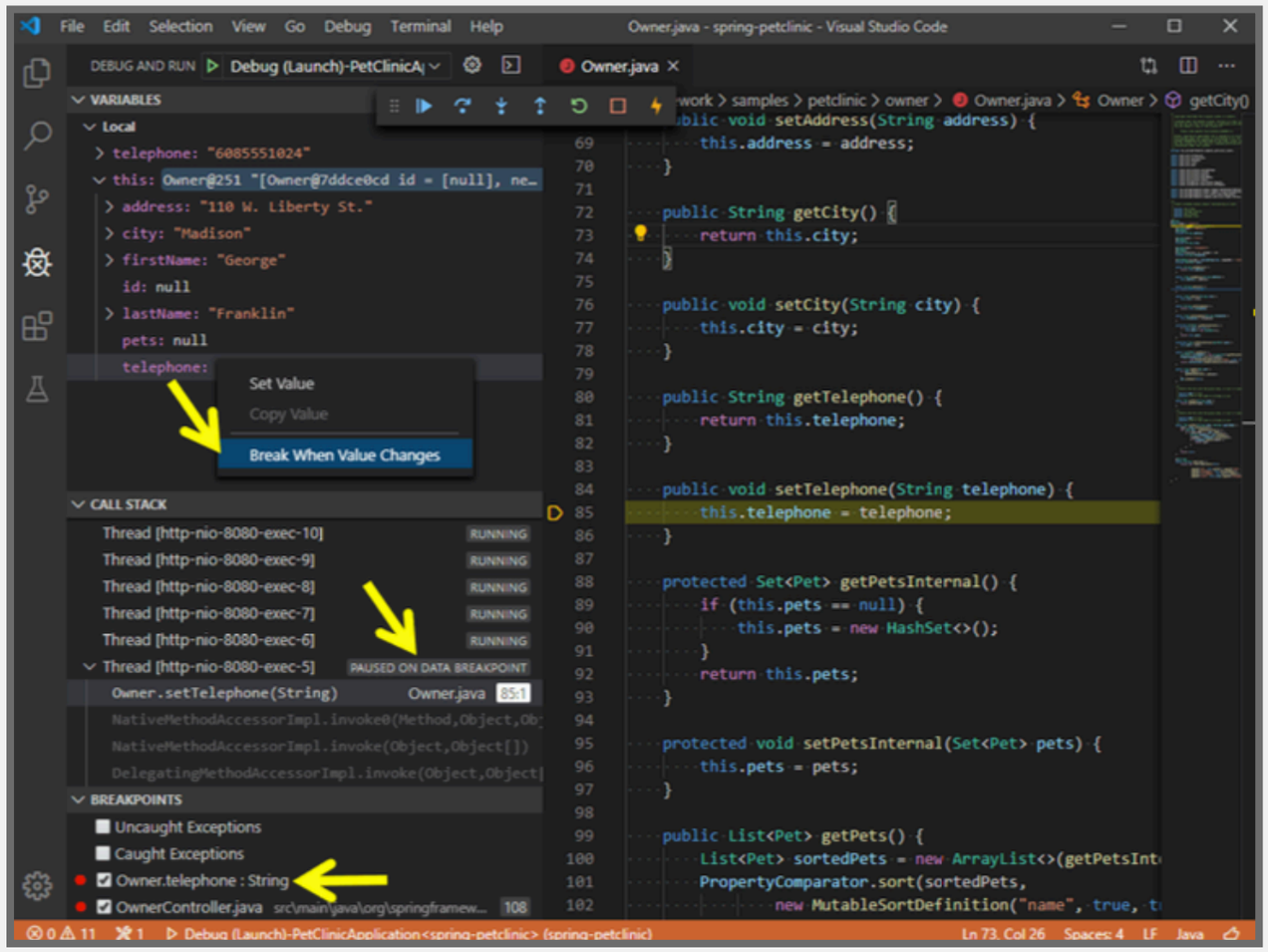
- Build it.
- Run it.
- Change it.
- Run it again.
- How did the behavior change?



Probes: Observe, Control, or "Lightly" Manipulate Execution



- print("this code is running!")
- Structured logging
- Debuggers
  - Breakpoint, eval, step through / step over
  - (Some tools even support remote debugging)
- Delete debugging
- Chrome Developer Tools



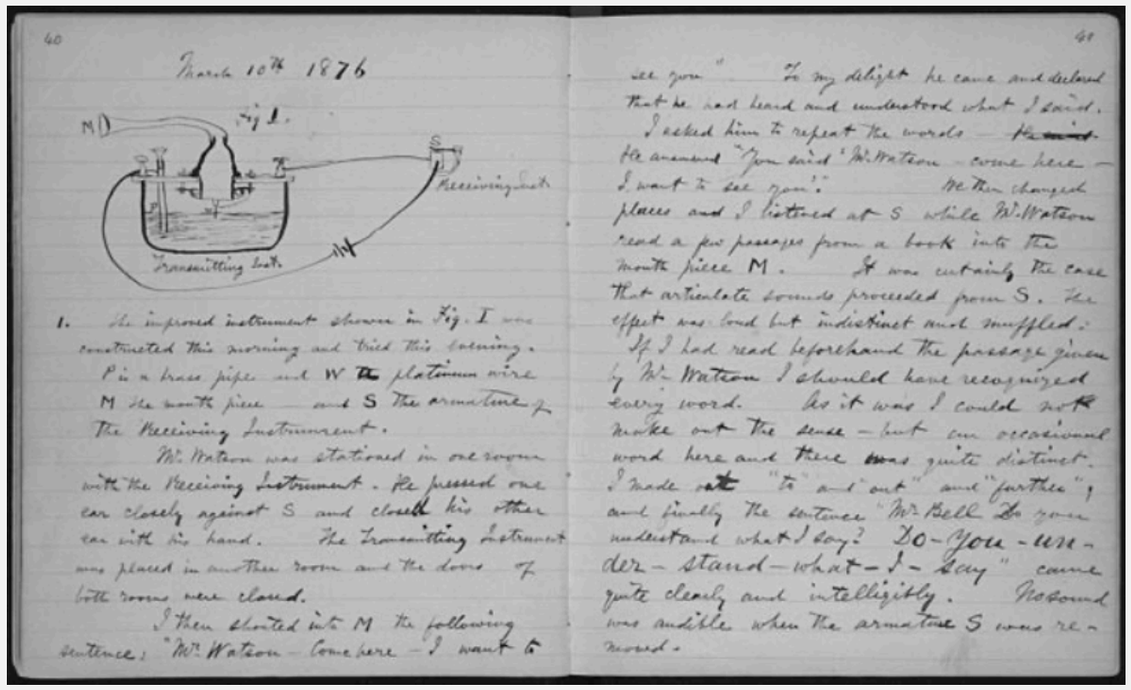
Step 0: Sanity Check Basic Model + Hypotheses



- *Confirm that you can build and run the code.*
  - Ideally both using the tests provided, and by hand.
- *Confirm that the code you are running is the code you built*
- *Confirm that you can make an externally visible change*
- *How? Where? Starting points:*
  - Run an existing test, change it
  - Write a new test
  - Change the code, write or rerun a test that should notice the change
- *Ask someone for help*

Document and Share Your Findings!

- Update README and docs
  - Or better: use a Developer Wiki
  - Use Mermaid for diagrams
- Screencast on Twitch
- Collaborate with others
- Include negative results, too!



Week 2 - Measurement & Metrics

What is Measurement?

- Measurement is the empirical, objective assignment of numbers, according to a rule derived from a model or theory, to attributes of objects or events with the intent of describing them. – Craner, Bond, “Software Engineering Metrics: What Do They Measure and How Do We Know?”

- A quantitatively expressed reduction of uncertainty based on one or more observations. – Hubbard, “How to Measure Anything ...”

Software Quality Metrics



- IEEE 1061 definition: “A software quality metric is a function whose inputs are software data and whose output is a single
  numerical value that can be interpreted as the degree to which the software possesses a given attribute that affects its quality.”

- Metrics have been proposed for many quality attributes; may define own metrics

What Software Qualities Do We Care About?



- Functionality (e.g., data integrity)
- Scalability
- Security
- Extensibility
- Bugginess
- Documentation
- Performance
- Installability
- Availability
- Consistency
- Portability
- Regulatory compliance

What Process Qualities Do We Care About?



- On-time release
- Development speed
- Meeting efficiency
- Conformance to processes
- Time spent on rework
- Reliability of predictions
- Fairness in decision making
- Number of builds
- Code review acceptance rate
- Regulatory compliance

- Measure time, costs, actions, resources, and quality of work packages; compare with predictions
- Use information from issue trackers, communication networks, team structures, etc...

What People Qualities Do We Care About?



- Developers
  - Maintainability
  - Performance
  - Employee satisfaction and well-being • Communication and collaboration
  - Efficiency and flow
  - Satisfaction with engineering system • Regulatory compliance

- Customers
  - Satisfaction
  - Ease of use
  - Feature usage
  - Regulatory compliance

Everything is Measurable

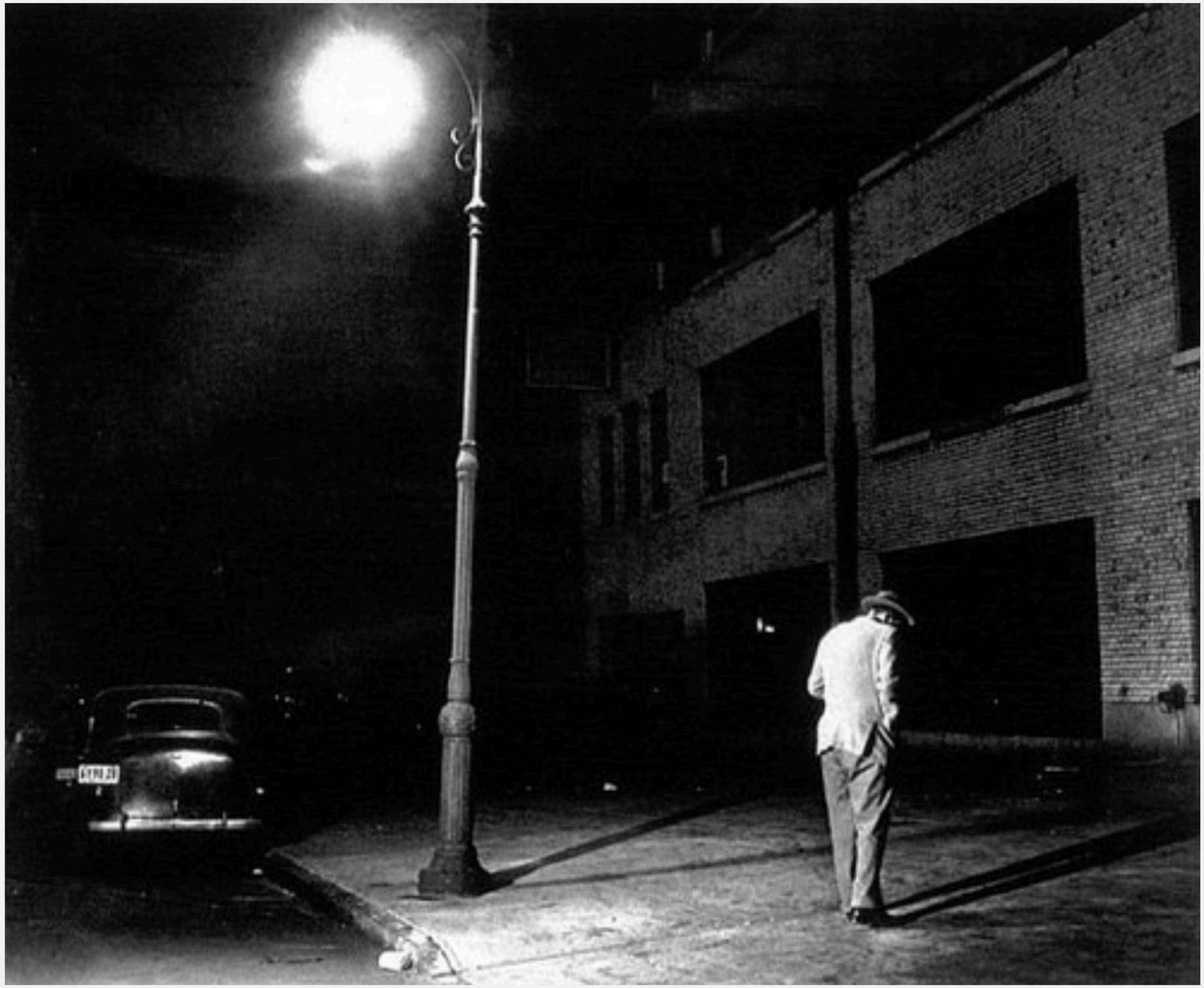
- If X is something we care about, then X, by definition, must be detectable.
  - How could we care about things like “quality,” “risk,” “security,” or “public image” if these things were totally undetectable, directly or indirectly?
  - If we have reason to care about some unknown quantity, it is because we think it corresponds to desirable or undesirable results in some way.
- If X is detectable, then it must be detectable in some amount.
  - If you can observe a thing at all, you can observe more of it or less of it 21
- If we can observe it in some amount, then it must be measurable.

Measurement for Decision Making



- Fund project?
- More testing?
- Fast enough? Secure enough?
- Code quality sufficient?
- Which feature to focus on?
- Developer bonus?
- Time and cost estimation? Predictions reliable?

The Streetlight Effect



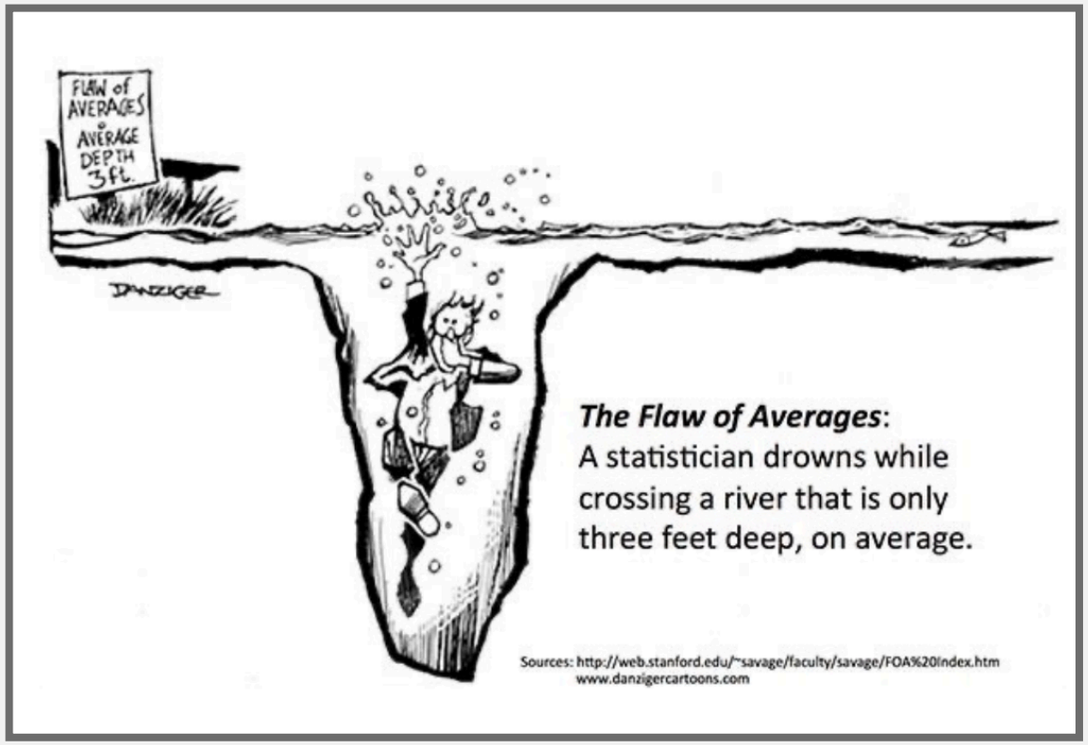
The Streetlight Effect



- A known observational bias.
- People tend to look for something only where it's easiest to do so.
- If you drop your keys at night, you'll tend to look for it under streetlights.

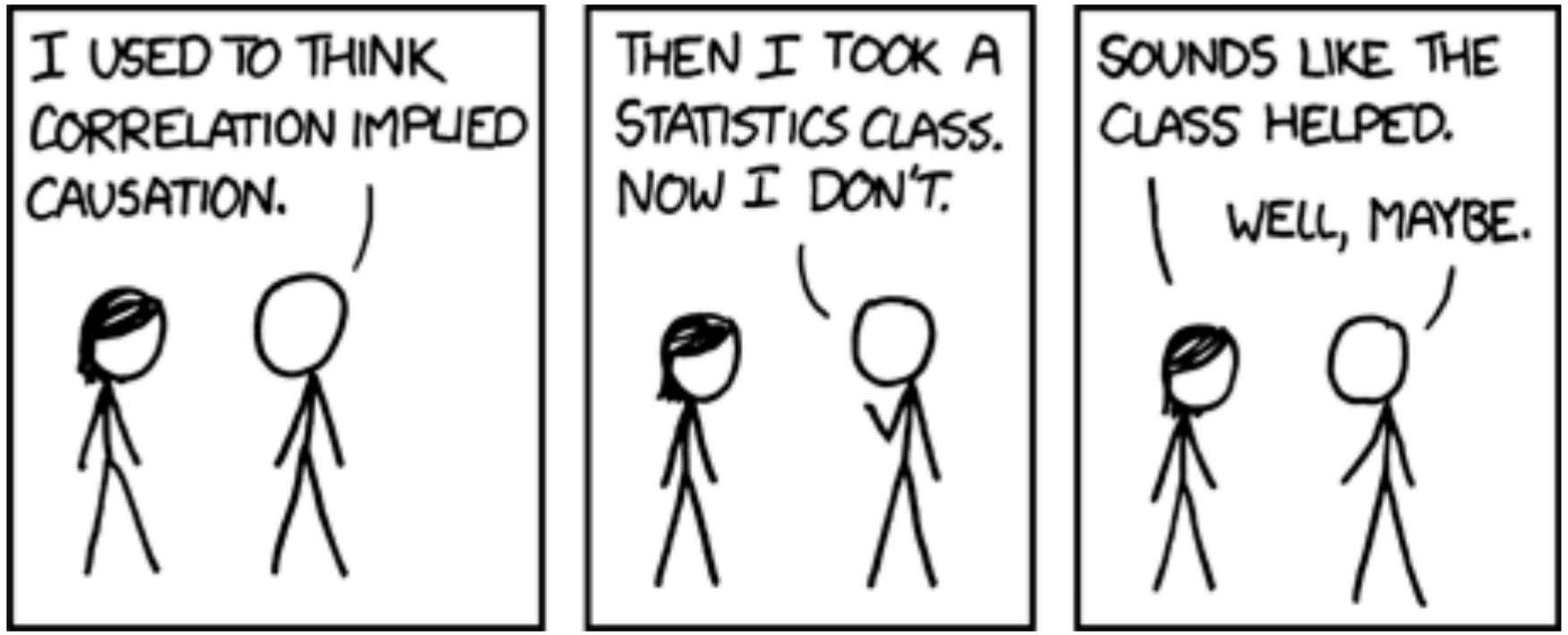
What could Possibly go Wrong?

- Bad statistics: A basic misunderstanding of measurement theory and what is being measured.
- Bad decisions: The incorrect use of measurement data, leading to unintended side effects.
- Bad incentives: Disregard for the human factors, or how the cultural change of taking measurements will affect people.



Making Inferences

http://xkcd.com/552/



- To infer causation:
  - Provide a theory (from domain knowledge, independent of data)
  - Show correlation
  - Demonstrate ability to predict new cases (replicate/validate)

Measurements Validity

- **Construct validity** – Are we measuring what we intended to measure?
- **Internal validity** – The extent to which the measurement can be used to explain some other characteristic of the entity being measured
- **External validity** – Concerns the generalization of the findings to contexts and environments, other than the one studied

Measurements Reliability



- Extent to which a measurement yields similar results when applied multiple times
- Goal is to reduce uncertainty, increase consistency
- Example: Performance
  - Time, memory usage
  - Cache misses, I/O operations, instruction execution count, etc.
- Law of large numbers
  - Taking multiple measurements to reduce error
- Trade-off with cost

McNamara Fallacy

- Measure whatever can be easily measured.
- Disregard that which cannot be measured easily.
- Presume that which cannot be measured easily is not important.
- Presume that which cannot be measured easily does not exist.

McNamara Fallacy

- There seems to be a general misunderstanding to the effect that a mathematical model cannot be undertaken until every constant and functional relationship is known to high accuracy. This often leads to the omission of admittedly highly significant factors (most of the “intangibles” influences on decisions) because these are unmeasured or unmeasurable. To omit such variables is equivalent to saying that they have zero effect... Probably the only value known to be wrong...

- J. W. Forrester, Industrial Dynamics, The MIT Press, 1961

Metrics & Incentives

- Goodhart's law: "When a measure becomes a target, it ceases to be a good measure."



Developer Productivity Myths



- Productivity is all about developer activity
- Productivity is only about individual performance
- One productivity metric can tell us everything
- Productivity measures are useful only for managers
- Productivity is only about engineering systems and developer tools

Summary



- Measurement is difficult but important for decision making
- Software metrics are easy to measure but hard to interpret,
  validity often not established
- Many metrics exist, often composed; pick or design suitable metrics if needed
- Careful in use: monitoring vs incentives
- Strategies beyond metrics



All Software Development Processes



Discuss the software that needs to be written

Write some code

Test the code to identify the defects

Debug to find causes of defects

Fix the defects

Let's Improve the Reliability of this Process



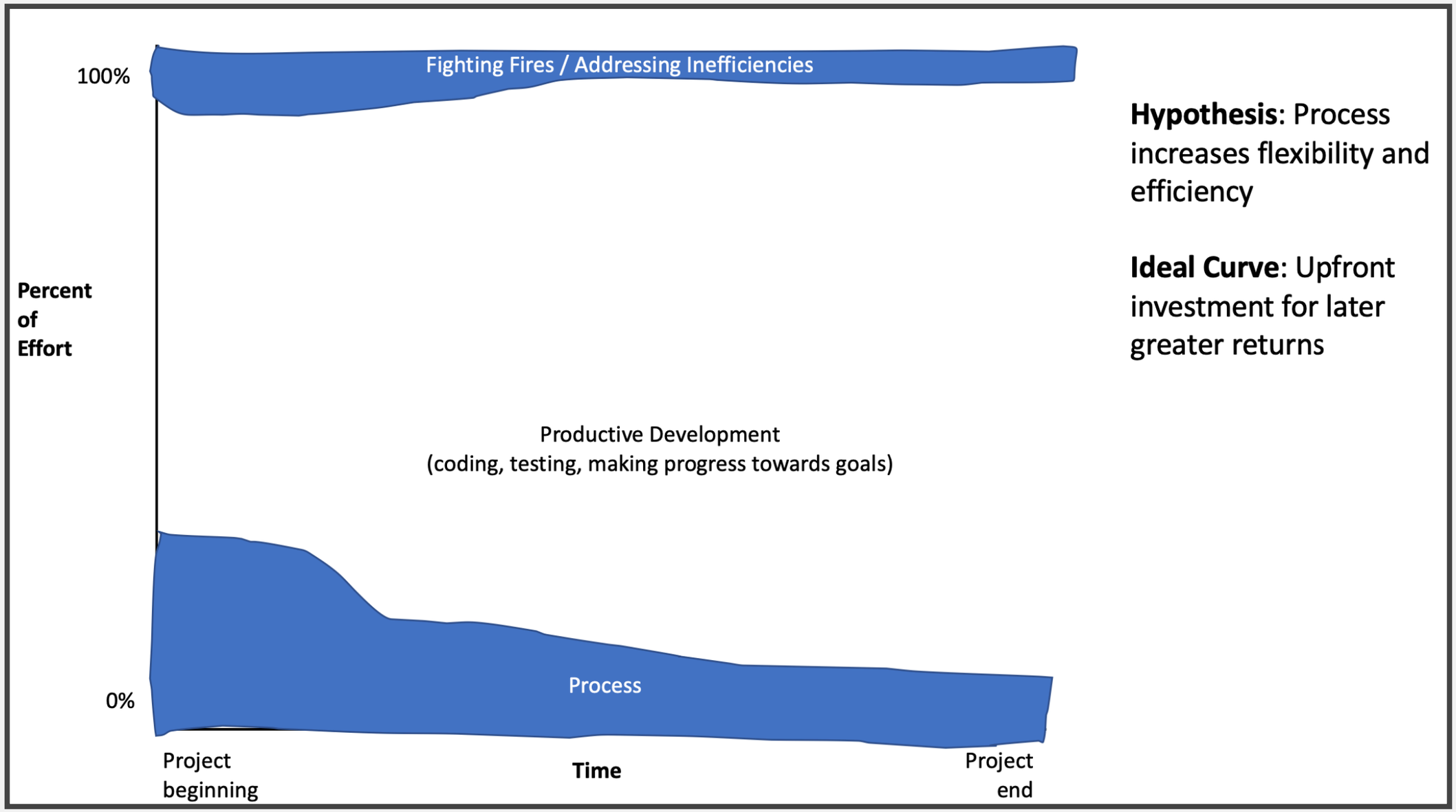
- Write down all requirements
  - Review requirements
  - Require approval for all changes to requirements
- Use version control for all changes
  - Review code
- Track all work items
  - Break down feature development into small tasks
  - Write down and monitor all reported bugs
- Hold regular, frequent status meetings
  - Plan and conduct quality assurance
  - Employ a DevOps framework to push code between developers and operations

Example Process Issues



- **Change Control:** Mid-project informal agreement to changes suggested by customer. Project scope expands 25-50%
- **Quality Assurance:** Late detection of requirements and design issues. Test-debug-reimplement cycle limits development of new features. Release with known defects.
- **Defect Tracking:** Bug reports collected informally. Bugs are overlooked.
- **System Integration:** Integration of independently developed components at the very end of the project. Interfaces out of sync.
- **Source Code Control:** Accidentally overwrote changes. Lost work.
- **Scheduling:** Late project. Developers asked to re-estimate work effort weekly.

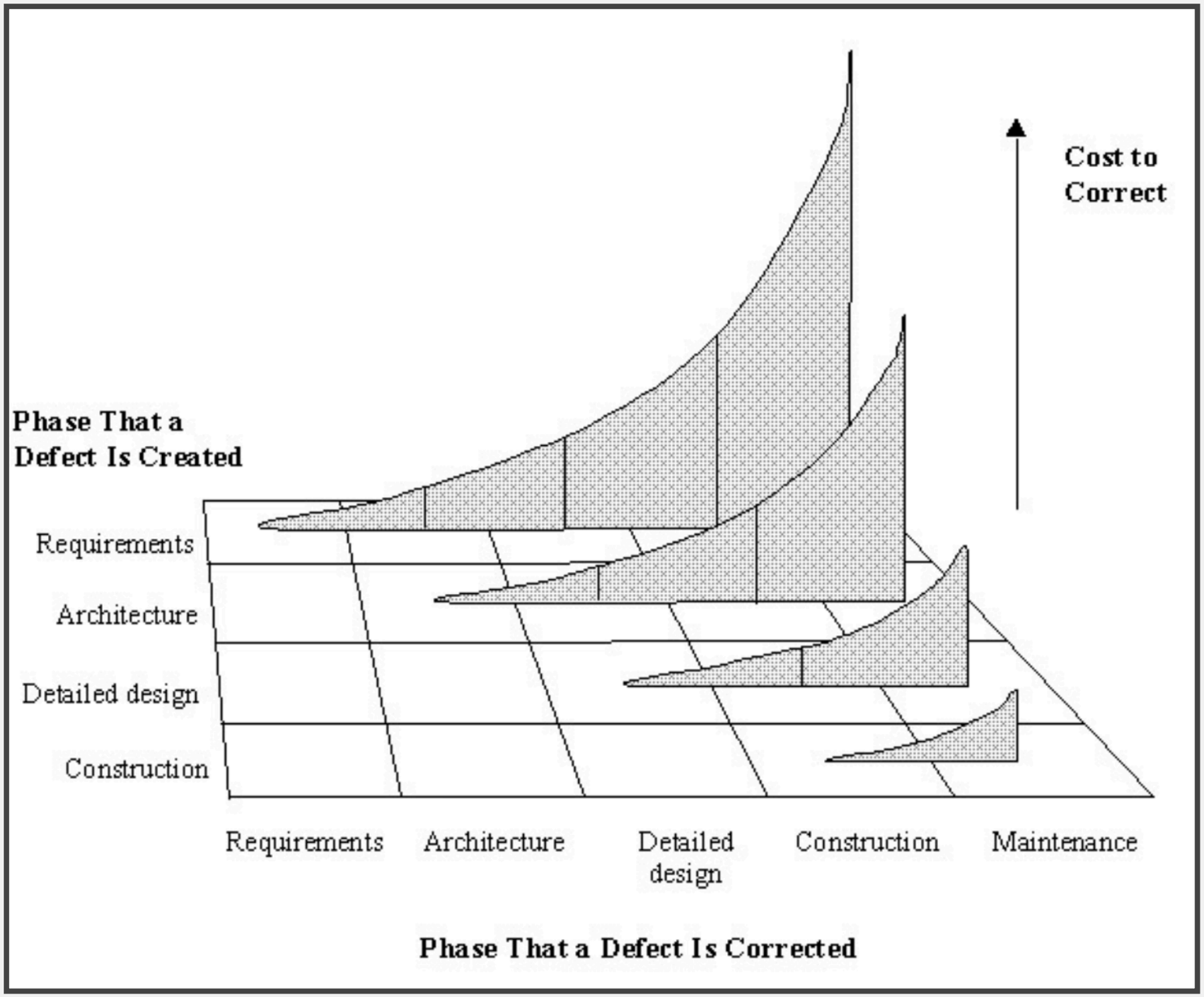
Effort Spent During the Process



Hypothesis: Process increases flexibility and efficiency

Ideal Curve: Upfront investment for later greater returns

Defect Correction Effort



Measuring Progress



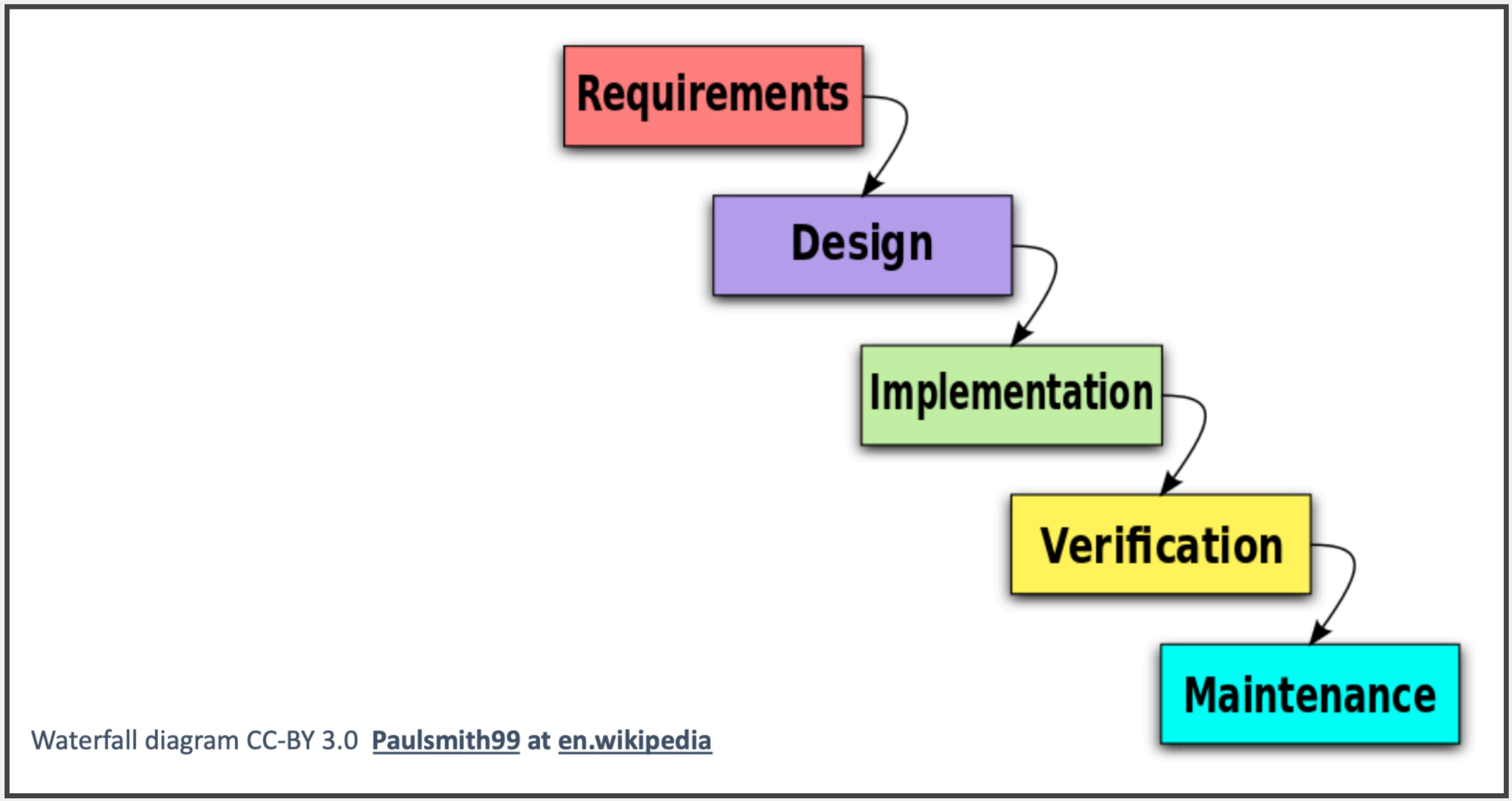
- “I’m almost done with the app. The frontend is almost fully implemented. The backend is fully finished except for the one stupid bug that keeps crashing the server. I only need to find the one stupid bug, but that can probably be done in an afternoon. We should be ready to release next week.”

Milestones and Deliverables Make Progress Observable



- **Milestone:** clear end point of a (sub)tasks
  - For project manager
  - Reports, prototypes, completed subprojects
    - “80% done“ is not a suitable mile stone
- **Deliverable:** Result for customer
  - Similar to a milestone, but for customers
  - Reports, prototypes, completed subsystems

Waterfall was the OG Software Process



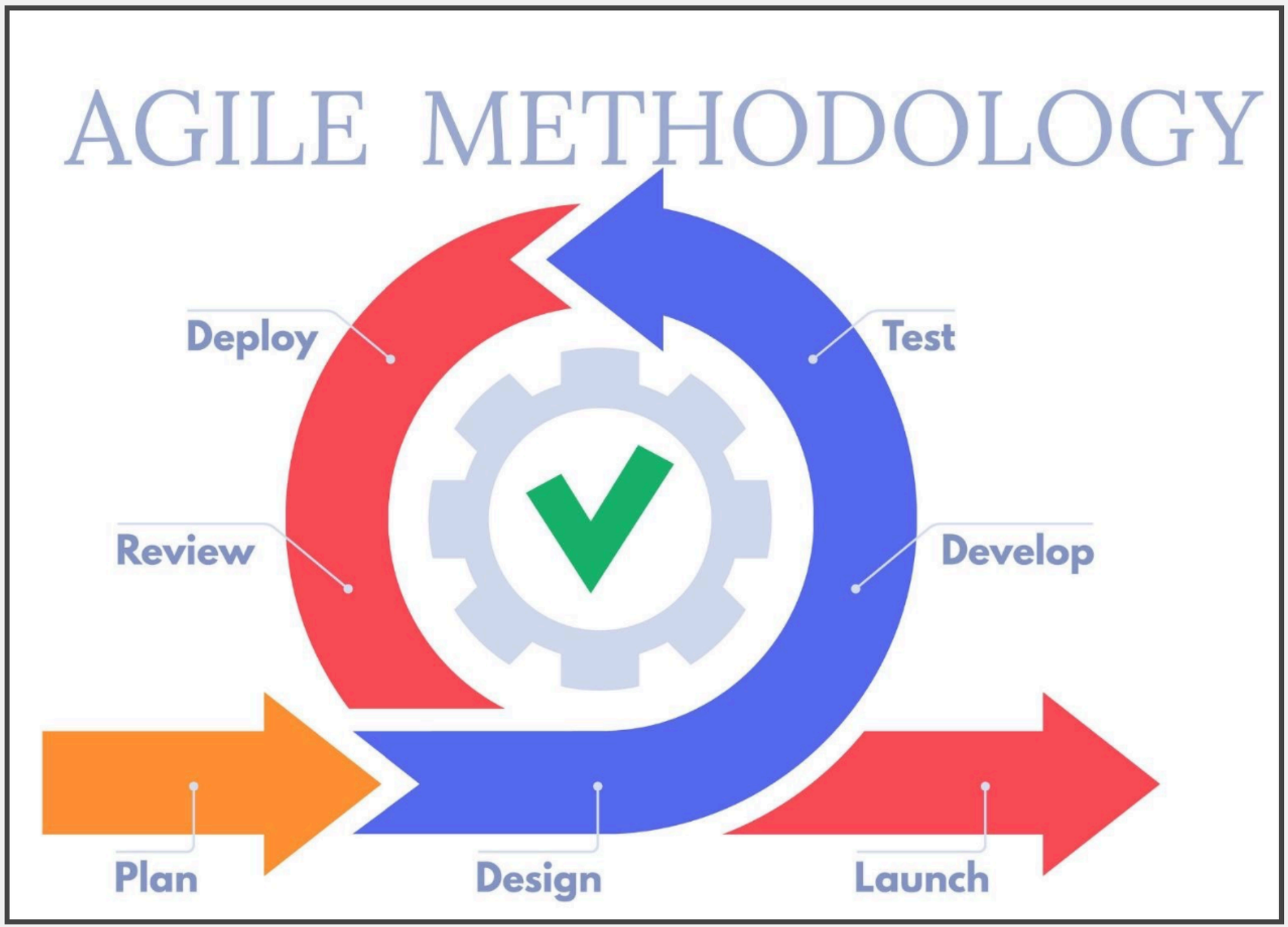
Waterfall diagram CC-BY 3.0 **Paulsmith99** at **en.wikipedia**

LEAN Production Adapts to Variable Demand


Taiichi Ohno

- Toyota Production System (TPS)
  - Build only what is needed, only when it is needed.
  - Use the “pull” system to avoid overproduction (Kanban)
  - Stop to fix problems, to get quality right from the start (Jidoka)
    - Workers are multi-skilled and understand the whole process; take ownership
- Lots of recent software buzzwords build on these ideas
  - Just-in-time, DevOps, Shift-Left

Now, Most Teams use some form of Agile Methods



Now, Most Teams use some form of Agile Methods

Agile software development

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

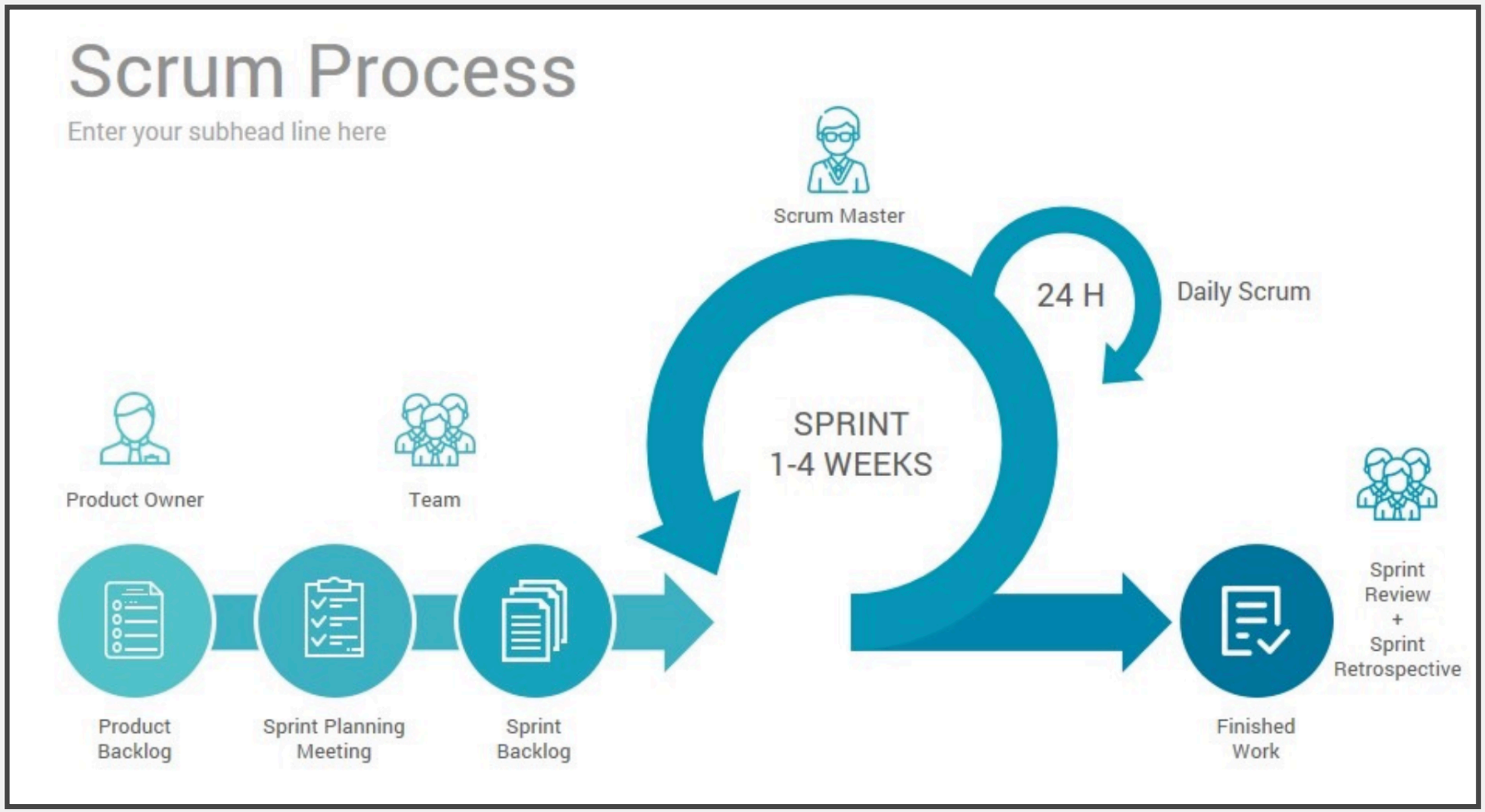
Manifesto for Agile Software Development (2001)



Core Concepts in Agile

Core concepts	Facets of agility in the literature
(1) Incremental design and iterative development	*Anticipating change by working iteratively – in short, delivery cycles – and thereby reducing the scope of the product to small increments to create opportunities for inspection; Creating change through incremental software design in response to change from what has been learned*
(2) Inspect and adapt cycles	*Anticipating change by instituting ceremonies for inspecting and adapting (i.e., learning from and creating change in response to discovered changes) the product increment (e.g., simplifying – "just enough" – design, testing software frequently) and the development process (e.g., updating work statuses, reevaluating team processes, reprioritizing requirements)*
(3) Working cooperatively/ Collaboratively/In close communication	*Anticipating change through recognising and predicting changes in one's environment; Creating change as a team by working together to respond to change from what has been learned collectively*
(4) Continuous customer involvement	*In addition to the cell above, centralising user requirements changes by working together with the customer to collectively identify and respond to change early through close customer involvement*

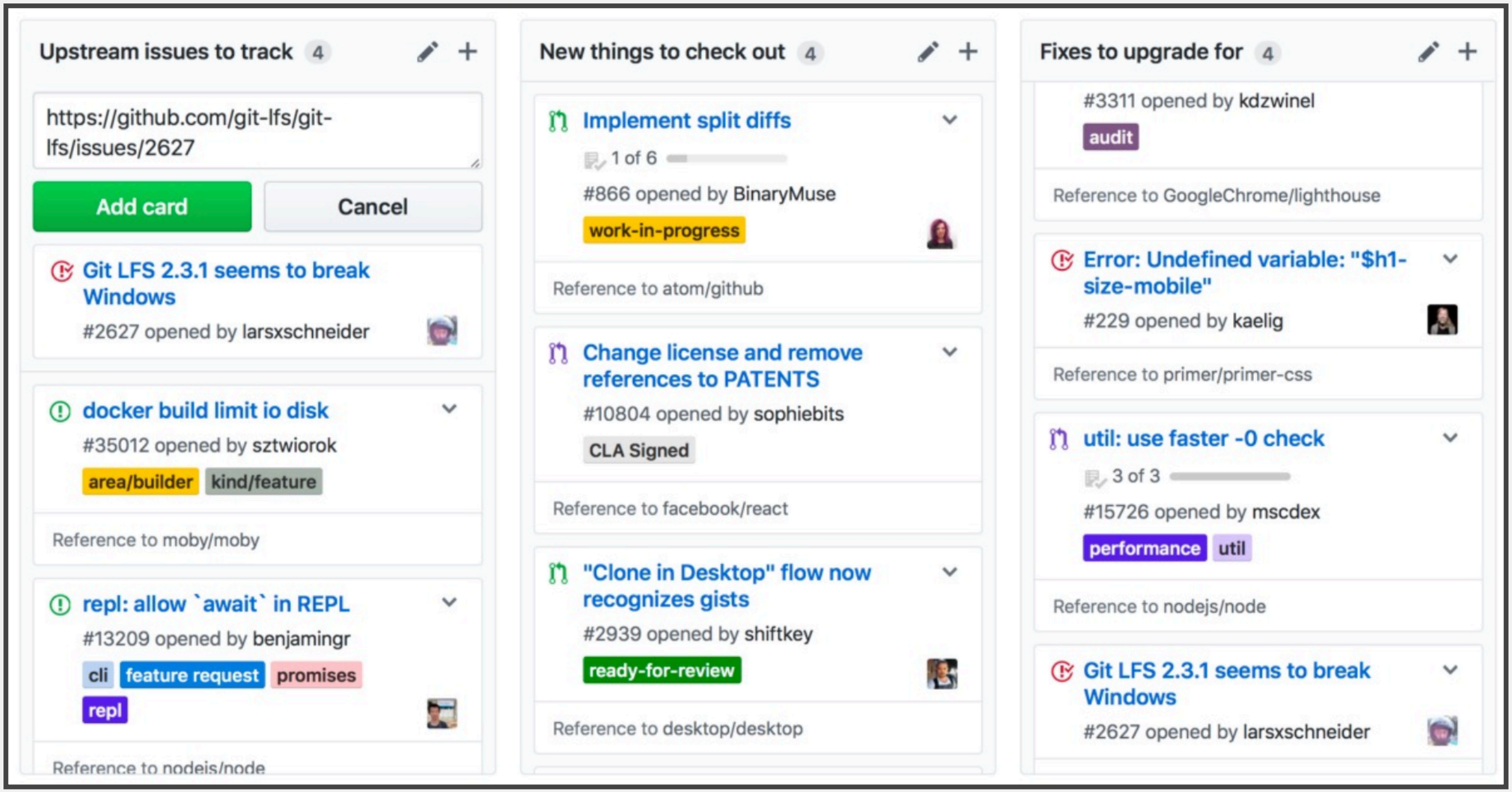
Elements of Scrum



Backlogs

- The product backlog is all the features for the product
- The sprint backlog is all the features that will be worked on for that sprint. These should be broken down into discrete tasks:
  - Fine-grained
  - Estimated
  - Assigned to individual team members
  - Acceptance criteria should be defined
- User Stories are often used

Kanban Boards



Scrum Meetings



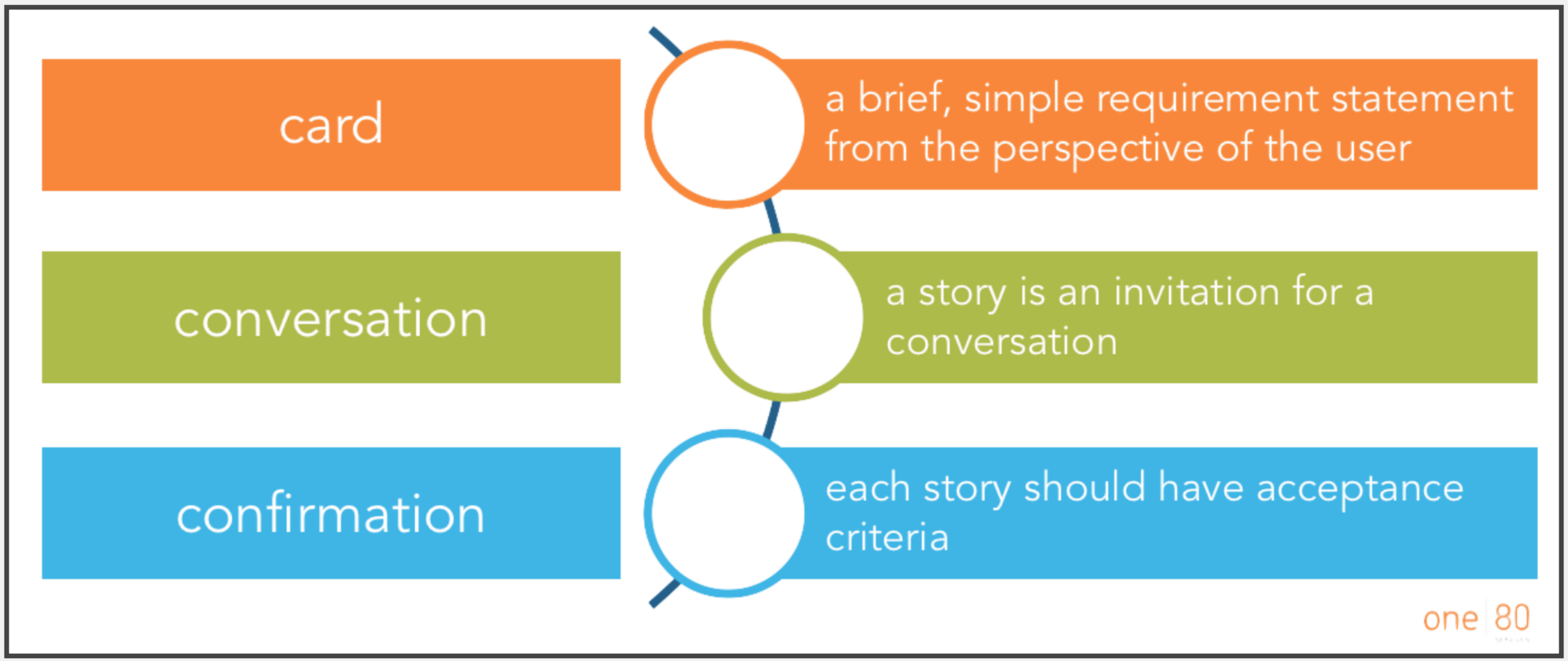
- Sprint Planning Meeting
  - Entire Team decides together what to tackle for that sprint

- • Daily Scrum Meeting
  - Quick Meeting to touch base on :
  - What have I done? What am I doing next? What am I stuck on/need help?

- Sprint Retrospective
  - Review sprint process

- Sprint Review Meeting
  - Review Product

User Stories



How to Evaluate a User Story



Independent



- Schedule in any order.
- Not overlapping in concept.
- Not always possible.



Negotiable

- Details to be negotiated during development.
- A good story captures the essence, not the details.



Valuable



- This story needs to have value to someone (hopefully the customer).
- Especially relevant to splitting up issues.



Estimable



- Helps keep the size small.
- Ensure we negotiated correctly.
- “Plans are nothing, planning is everything” - Dwight D. Eisenhower

Small



- Can be written on a 3x5 card.
- At most two person-weeks of work.
- Too big === unable to estimate



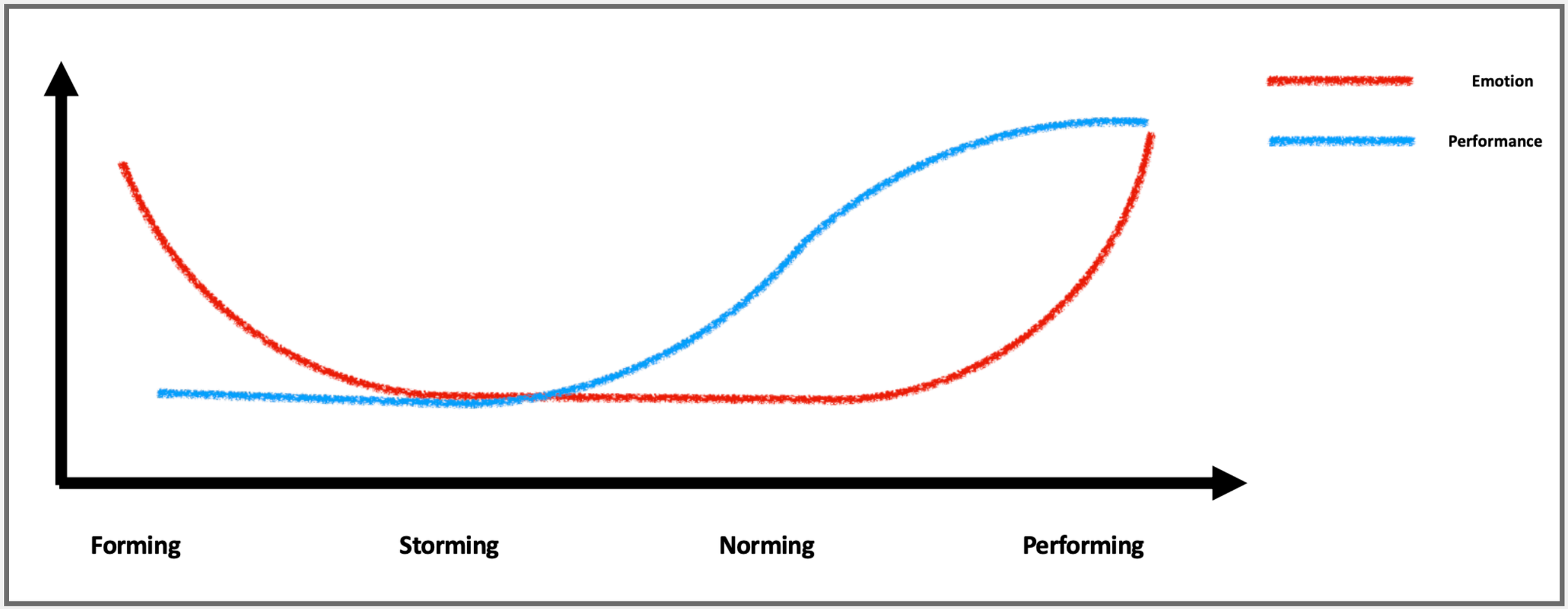
Testable



- Ensures understanding of task
- We know when we can mark task “Done”
- Unable to test === I do not understand it



Stages of Team Formation



Norming

- When working with someone who is remote, how do you like to work together?
- How do you manage your time when you get busy with a lot of tasks?
- How do you feel about chatting by text message, audio call, video call?
    - Exchange phone numbers with your project partner(s) in case your Internet goes out and you still want to work on the project together.
- Negotiate when you can work on the project together outside of class.
- Have you had a positive prior teaming experience?
    - How often did your team meet?
    - Did your team have a leader? If yes, what did that leader do?
    - What was your role on the team?
    - How well did you get along with your teammates related to work, or related to non-work?

Establish Communication Patterns



- Asana, Trello, Microsoft Projects, …
- Github Wiki, Google Docs, Notion, ...
- Github Issues, Jira, …
- Email, Slack, Facebook groups, …
- Zoom, Microsoft Teams, Skype, Phone call, ...
- Face-to-face meetings

Check Out Other Projects



Communication

- Forums: Discuss implementations, research, etc. https://discuss.pytorch.org
- GitHub Issues: Bug reports, feature requests, install issues, RFCs, thoughts, etc.
- Slack: The PyTorch Slack hosts a primary audience of moderate to experienced PyTorch users and developers for general chat, online discussions, collaboration, etc. If you are a beginner looking for help, the primary medium is PyTorch Forums. If you need a slack invite, please fill this form: https://goo.gl/forms/PP1AGvNHpSaJP8to1
- Newsletter: No-noise, a one-way email newsletter with important announcements about PyTorch. You can sign-up here: https://eepurl.com/cbG0rv
- Facebook Page: Important announcements about PyTorch. https://www.facebook.com/pytorch
- For brand guidelines, please visit our website at pytorch.org

Communication Expectation

- Quality of service guarantee
  - How soon will you get back to your teammates?
  - Weekend? Evening?
- Emergency
  - Tag w/ 911
  - Notify everyone with @channel

How to Run a Meeting



- The Three Rules of Running a Meeting
  - Set the Agenda
  - Start on Time. End on Time.
  - End with Action Items (and share them - Github Issues, Meeting Notes, ...)

How to Run a Meeting



- The Three Rules of Running a Meeting
  - Set the Agenda
  - Start on Time. End on Time.
  - End with Action Items (and share them - Github Issues, Meeting Notes, ...)

Writing Useful Github Issues

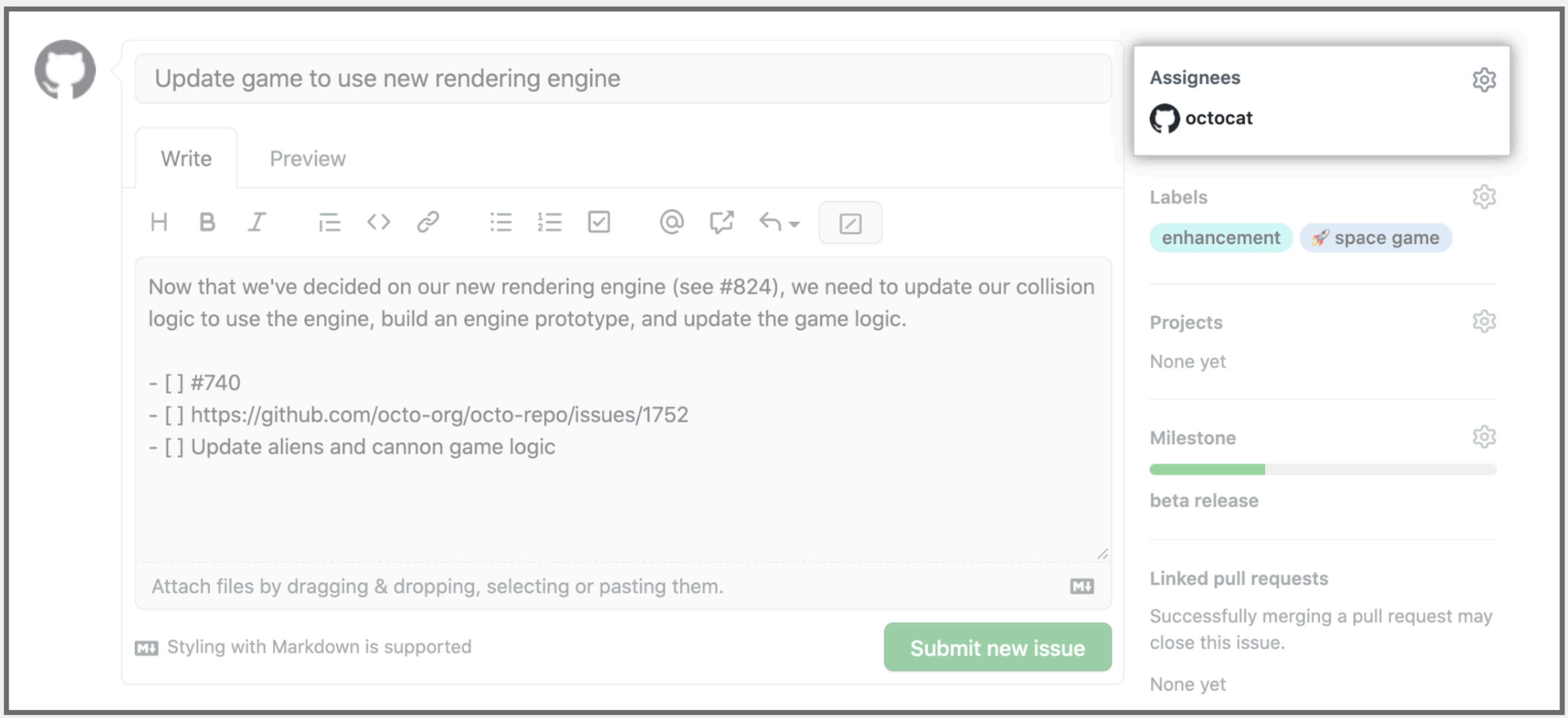


Writing Useful Github Issues



- Issue should include
  - Context: explain the conditions which led you to write the issue
  - Problem or idea: the context should lead to something
  - Previous attempts to solve
  - Solution or next step (if possible)
- Be specific!
  - Include environment settings, versions, error messages, code examples when necessary

@Mention or Assign Appropriate People



Use Labels

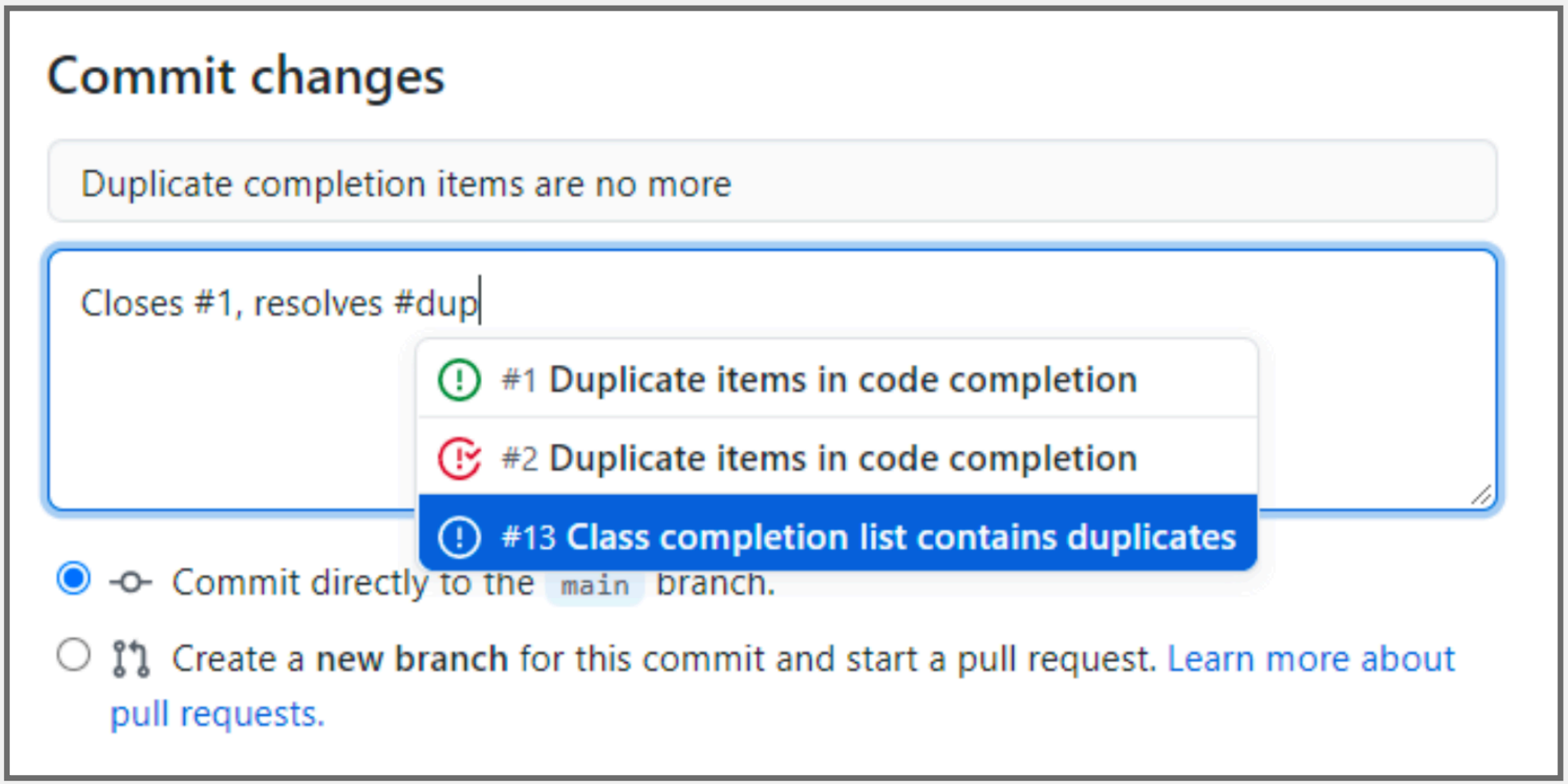


- Break the project down by areas of responsibility
- Mark non-triaged issues
- Isolate issues that await additional information from the reporter
- Example:
  - Bug / Duplicate / Documentation / Help Wanted / Invalid / Enhancement
  - status: wip, status: ready to implement, status: needs discussion

Don’t Forget to Follow Up and Close Issues



- closes/resolves #issue_number



How to Write Good Pull Requests



```
## What?

## Why?

## How?

## Testing?

## Screenshots (optional)

## Anything Else?
```

How to Write Good Pull Requests



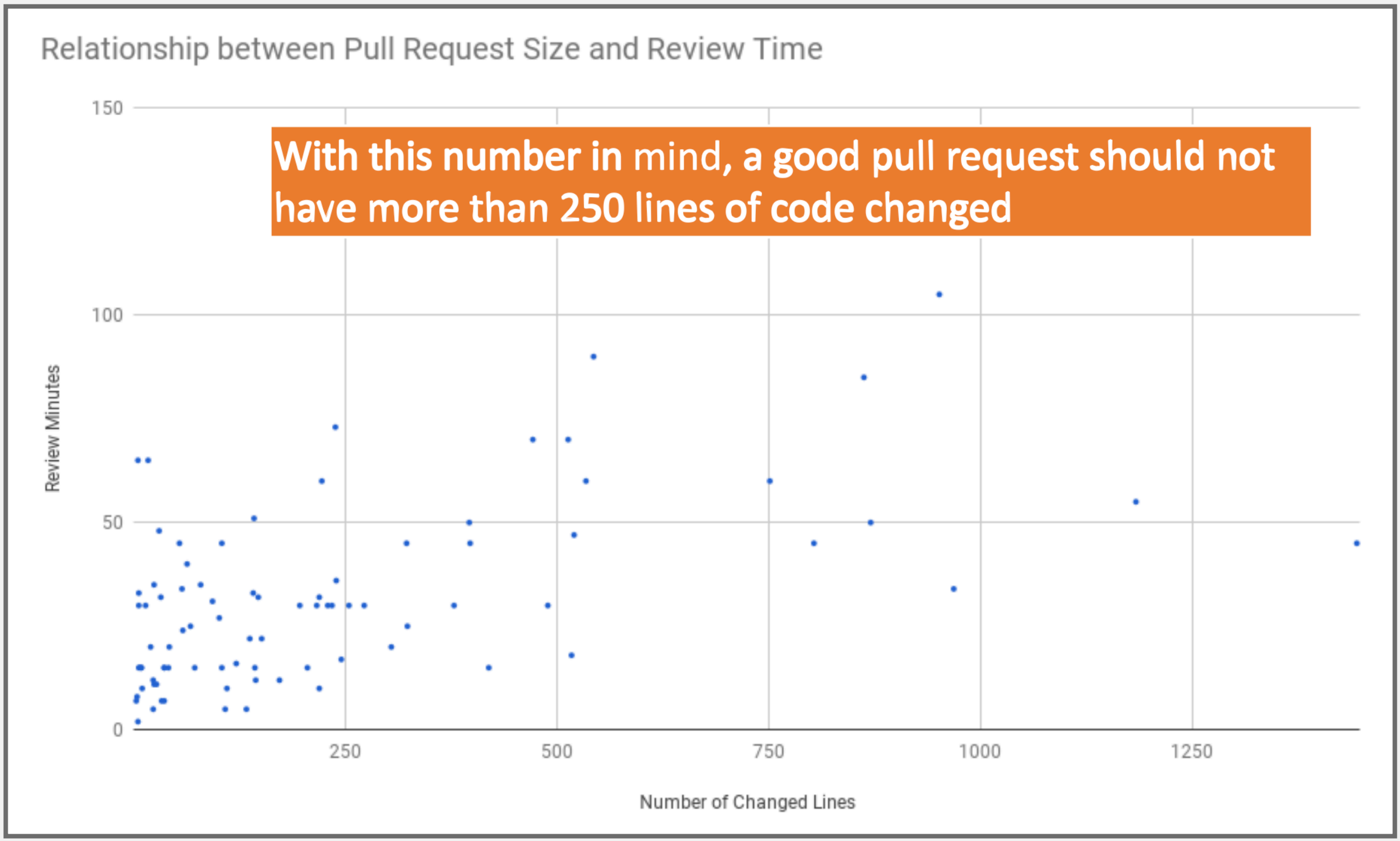
```
## What?
I've added support for authentication to implement Key Result 2 of OKR1. It includes model, table,
controller and test. For more background, see ticket
#JIRA-123.
## Why?
These changes complete the user login and account creation experience. See #JIRA-123 for more
information.
## How?
This includes a migration, model and controller for user authentication. I'm using Devise to do the
heavy lifting. I ran Devise migrations and those are included here.
## Testing?
I've added coverage for testing all new methods. I used Faker for a few random user emails and
names.
## Screenshots (optional)
0
## Anything Else?
Let's consider using a 3rd party authentication provider for this, to offload MFA and other
considerations as they arise and as the privacy landscape evolves. AWS Cognito is a good option, so
is Firebase. I'm happy to start researching this path. Let's also consider breaking this out into
its own service. We can then re-use it or share the accounts with other apps in the future.
```

How to Write Good Pull Requests



- Remember that anyone (in the company) could be reading your PR
- Be explicit about what/when feedback you want
- @mention individuals that you specifically want to involve in the discussion, and mention why.
  - “/cc @jesseplusplus for clarification on this logic”

Keep your PRs Small



Offer Useful Feedback



- If you disagree strongly, consider giving it a few minutes before responding; think before you react.
- Ask, don't tell. ("What do you think about trying...?" rather than "Don't do...")
- Explain your reasons why code should be changed. (Not in line with the style guide? A personal preference?)
- Be humble. ("I'm not sure, let's try...")
- Avoid hyperbole. ("NEVER do...")
- Be aware of negative bias with online communication.

Importance of Documentation

No matter the format, documentation is important

Building on top of others' work in a community-like way can be an accelerator, both in open source and in companies. Documentation often signals if a repository is reliable to reuse code from, or if it's an active project to contribute to. What signs do developers look for?

In both open source projects and enterprises, developers see about

50%

productivity boost with easy-to-source documentation

What the data shows: At work, developers consider documentation trustworthy when it is up-to-date (e.g., looking at time-stamps) and has a high number of upvotes from others. Open source projects use READMEs, contribution guidelines, and GitHub Issues, to elevate the quality of any project, and to share information that makes them more attractive to new contributors. Enterprises can adopt the same best practices to achieve similar success.

In both environments, developers see about a 50% productivity boost when documentation is up-to-date, detailed, reliable, and comes in different formats (e.g. articles, videos, forums).

Using the data: Review the documentation your team consumes: When was the last time it was updated? Can everyone on your team improve the documentation? Check this frequently to stay on track.

Types of Documentation

Knowledge Type	Description (Excerpt)
Functionality and Behavior	Describes what the API does (or does not do) in terms of functionality or features. Describes what happens when the API is used (a field value is set, or a method is called).
Concepts	Explains the meaning of terms used to name or describe an API element, or describes design or domain concepts used or implemented by the API.
Directives	Specifies what users are allowed / not allowed to do with the API element. Directives are clear contracts.
Purpose and Rationale	Explains the purpose of providing an element or the rationale of a certain design decision. Typically, this is information that answers a "why" question: Why is this element provided by the API? Why is this designed this way? Why would we want to use this?
Quality Attributes and Internal Aspects	Describes quality attributes of the API, also known as non-functional requirements, for example, the performance implications. Also applies to information about the API's internal implementation that is only indirectly related to its observable behavior.
Control-Flow	Describes how the API (or the framework) manages the flow of control, for example by stating what events cause a certain callback to be triggered, or by listing the order in which API methods will be automatically called by the framework itself.
Structure	Describes the internal organization of a compound element (e.g. important classes, fields, or methods), information about type hierarchies, or how elements are related to each other.
Patterns	Describes how to accomplish specific outcomes with the API, for example, how to implement a certain scenario, how the behavior of an element can be customized, etc.
Code **Examples**	Provides code examples of how to use and combine elements to implement certain functionality or design outcomes.
Environment	Describes aspects related to the environment in which the API is used, but not the API directly, e.g., compatibility issues, differences between versions, or licensing information.
References	Includes any pointer to external documents, either in the form of hyperlinks, tagged "see also" reference, or mentions of other documents (such as standards or manuals).
Non-information	A section of documentation containing any complete sentence or self-contained fragment of text that provides only uninformative boilerplate text.

Maalej, W., & Robillard, M. P. (2013). Patterns of knowledge in API reference documentation. IEEE Transactions on Software Engineering, 39(9), 1264-1282.

Types of Documentation

Knowledge Type	Description (Excerpt)
Functionality and Behavior	Describes what the API does (or does not do) in terms of functionality or features. Describes what happens when the API is used (a field value is set, or a method is called).
Concepts	Explains the meaning of terms used to name or describe an API element, or describes design or domain concepts used or implemented by the API.
Directives	Specifies what users are allowed / not allowed to do with the API element. Directives are clear contracts.
Purpose and Rationale	Explains the purpose of providing an element or the rationale of a certain design decision. Typically, this is information that answers a "why" question: Why is this element provided by the API? Why is this designed this way? Why would we want to use this?
Quality Attributes and Internal Aspects	Describes quality attributes of the API, also known as non-functional requirements, for example, the performance implications. Also applies to information about the API's internal implementation that is only indirectly related to its observable behavior.
Control-Flow	Describes how the API (or the framework) manages the flow of control, for example by stating what events cause a certain callback to be triggered, or by listing the order in which API methods will be automatically called by the framework itself.
Structure	Describes the internal organization of a compound element (e.g. important classes, fields, or methods), information about type hierarchies, or how elements are related to each other.
Patterns	Describes how to accomplish specific outcomes with the API, for example, how to implement a certain scenario, how the behavior of an element can be customized, etc.
Code **Examples**	Provides code examples of how to use and combine elements to implement certain functionality or design outcomes.
Environment	Describes aspects related to the environment in which the API is used, but not the API directly, e.g., compatibility issues, differences between versions, or licensing information.
References	Includes any pointer to external documents, either in the form of hyperlinks, tagged "see also" reference, or mentions of other documents (such as standards or manuals).
Non-information	A section of documentation containing any complete sentence or self-contained fragment of text that provides only uninformative boilerplate text.

Maalej, W., & Robillard, M. P. (2013). Patterns of knowledge in API reference documentation. IEEE Transactions on Software Engineering, 39(9), 1264-1282.

Know Your Audience



- Internal document for your team (e.g., meeting note)
- Documentation for project contributors
- Documentation for non-developer collaborators (e.g., UX researchers)
- Documentation for developer users
- Documentation for clients with no software knowldge
- User manual for end users

How to Ask Questions

New To Coding. Can anyone assist me?

Asked 7 years, 1 month ago Modified 7 years, 1 month ago Viewed 47 times

I am trying to make a word counter and I just cant seem to get it. Can anyone help?

-4

```
import re
print("Welcome To This Software Made By Aaron!")
word = raw_input("Enter Your Words: ")
Check = 0
Right = 0
Length = len(word)
while True:
    if Right == 1:
        if Length < Check:
            Check = Check + 1
            print(Check)
    if Length == Check:
        Right = 1

print("Your Word Count Is " +Check)
```



Make it Easy for People to Help You

- I am trying to ___, so that I can ___. I am running into ___.
  I have looked at ___ and tried ___.
- + I'm using this tech stack: ___.
- + I'm getting this error/result: ___.
- + I think the problem could be ___.

Conflict Resolution

- Your goal: Find a solution to the problem and move forward.
  - As a smart person on ”TedLasso” once said,“Fight forward,not back.”
- Make sure that everybody works from the same set of facts.
- Establish ground rules for your team’s discussion.
  - Talk about how the situation made you feel.Never presume anything about anyone else.
- Remain calm and rational. If you feel triggered or threatened, extract yourself from the situation, wait an hour to chill out, and then try again.
- If you reach an impasse, talk to your team leader.
- If your team remains in conflict, escalate to Dr. Moran.
  - I can help to mediate

Week 4- Software Testing



What is Testing Good For?



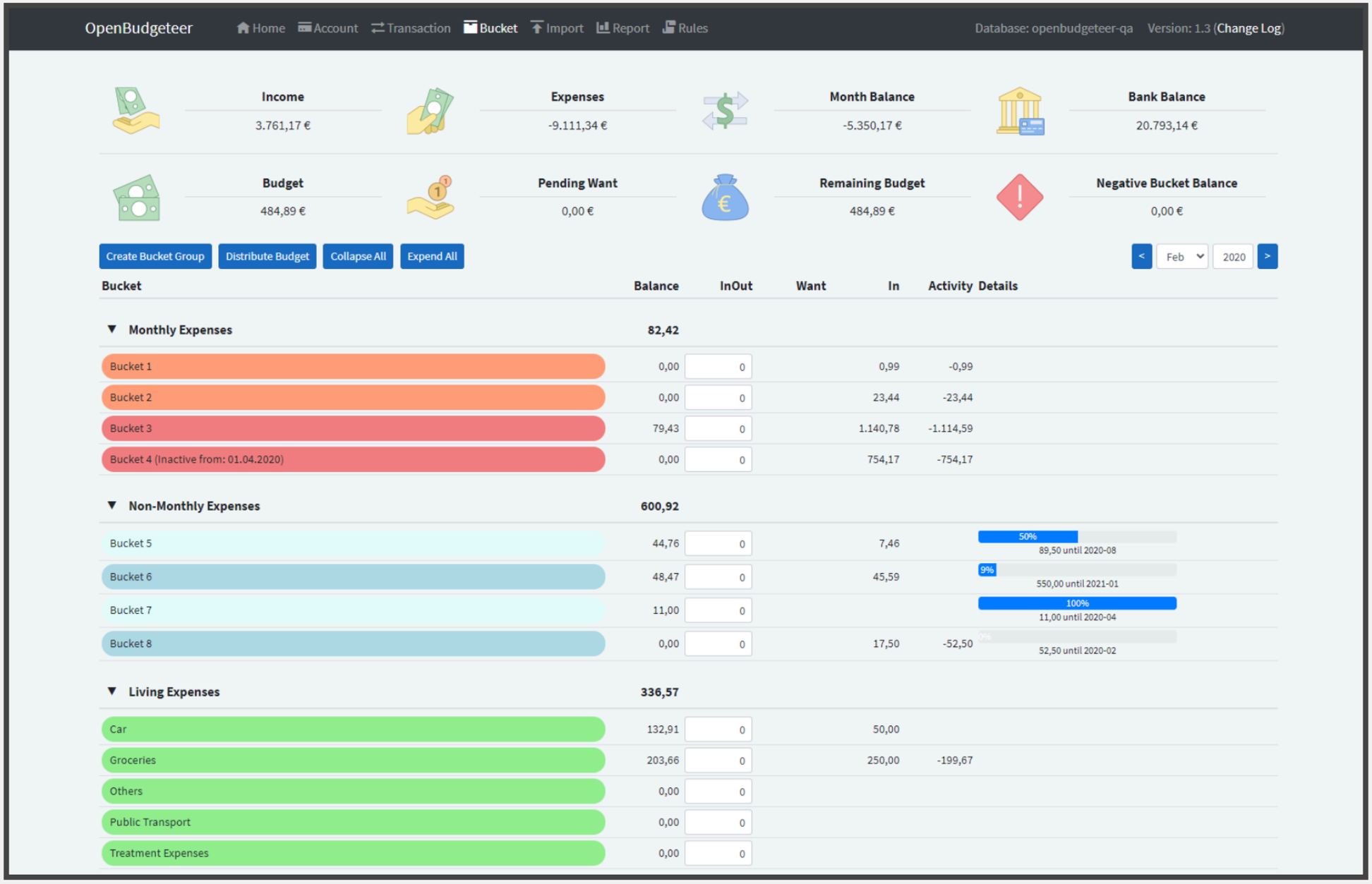
- What is testing?
  - Execution of code on sample inputs in a controlled environment
- Principle goals:
  - Validation: program meets requirements, including quality attributes.
  - Defect testing: reveal failures.

What is Testing Good For?



- Why should we test? What does testing achieve?
  - What does testing not achieve?
- When should we test?
  - And where should we run the tests?
- What should we test?
  - What CAN we test? (Software quality attributes)
- How should we test?
  - How many ways can you test the sort() function?
- How good are our tests?
  - How to measure test quality?

What Makes a Good Test?



https://github.com/TheAxelander/OpenBudgeteer

Why Write Tests at All?



- [Low bar] Ensure that our software meets requirements, is correct, etc.
- Preventing bugs or quality degradations from being accidentally introduced in the future -> *Regression Testing*
- Helps uncover unexpected behaviors that can't be identified by reading source code
- Increased confidence in changes ("will I break the internet with this commit?")
- Bridges the gap between a declarative view of the system (i.e., requirements) and an imperative view (i.e., implementation) by means of redundancy.
- Tests are executable documentation; increases code maintainability
- Forces writing testable code <-> checks software design

Testing Levels



- Unit testing
  - Code level, E.g. is a function implemented correctly?
  - Does not require setting up a complex environment
- Integration testing
  - Do components interact correctly? E.g. a feature that cuts across client and server.
  - Usually requires some environment setup, but can abstract/mock out other components that are not being tested (e.g. network)
- System testing
  - Validating the whole system end-to-end (E2E)
  - Requires complete deployment in a staging area, but fake data
- Testing in production
  - Real data but more risks

What are the Limitations of Testing?



- "*Testing shows the presence, not the absence of bugs.*" - Edsger W. Dijkstra
- Testing doesn't really give any formal assurances
- Writing tests is hard, time consuming
- Knowing if your tests are good enough is not obvious
- Executing tests can be expensive, especially as software complexity and configuration space grows
  - Full test suite for a single large app can take several days to run

Test Oracles



- ”Oracles” are mechanisms that tell you when program execution seems abnormal or unexpected
- E.g. assert, segfault, exception
- Other examples: performance threshold, memory footprint, address sanitizer

Test Oracles



- Obvious in some applications (e.g. “sort()”) but more challenging in others (e.g. “encrypt()” or UI-based tests)

- Lack of good oracles can limit the scalability of testing. Easy to generate lots of input data, but not easy to validate if output (or other program behavior) is correct.

- Fortunately, we have some tricks.

Differential Testing



- If you have two implementations of the same specification, then their output should match on all inputs.
  - E.g. `mergeSort(x).equals(bubbleSort(x))` -> should always be true
  - Special case of a property test, with a free oracle.

- If a differential test fails, at least one of the two implementations is wrong.
  - But which one?
  - If you have N>2 implementations, run them all and compare. Majority wins (the odd one out is buggy).

- Differential testing works well when testing programs that implement standard specifications such as compilers, browsers, SQL engines, XML/JSON parsers, media players, etc.
  - Not feasible in general e.g. for UCF's custom grad application system.

Regression Testing



- Differential testing through time (or versions, say V1 and V2).
- Assuming V1 and V2 don't add a new feature or fix a known bug, then f(x) in V1 should give the same result as f(x) in V2.
- *Key Idea:* Assume the current version is correct. Run program on current version and log output. Compare all future versions to that output.

Test Driven Development



- Tests first!
- Popular agile technique
- Write tests as specifications before code
- Never write code without a failing test

- Claims:
  - Design approach toward testable design
  - Think about interfaces first
  - Avoid unneeded code
  - Higher product quality
  - Higher test suite quality
  - Higher overall productivity

Common Bar for Contributions



Chromium

- **Changes should include corresponding tests.** Automated testing is at the heart of how we move forward as a project. All changes should include corresponding tests so we can ensure that there is good coverage for code and that future changes will be less likely to regress functionality. Protect your code with tests!

Firefox

Testing Policy

Everything that lands in mozilla-central includes automated tests by default. Every commit has tests that cover every major piece of functionality and expected input conditions.

Docker

Conventions

Fork the repo and make changes on your fork in a feature branch:

- If it's a bugfix branch, name it XXX-something where XXX is the number of the issue
- If it's a feature branch, create an enhancement issue to announce your intentions, and name it XXX-something where XXX is the number of the issue.

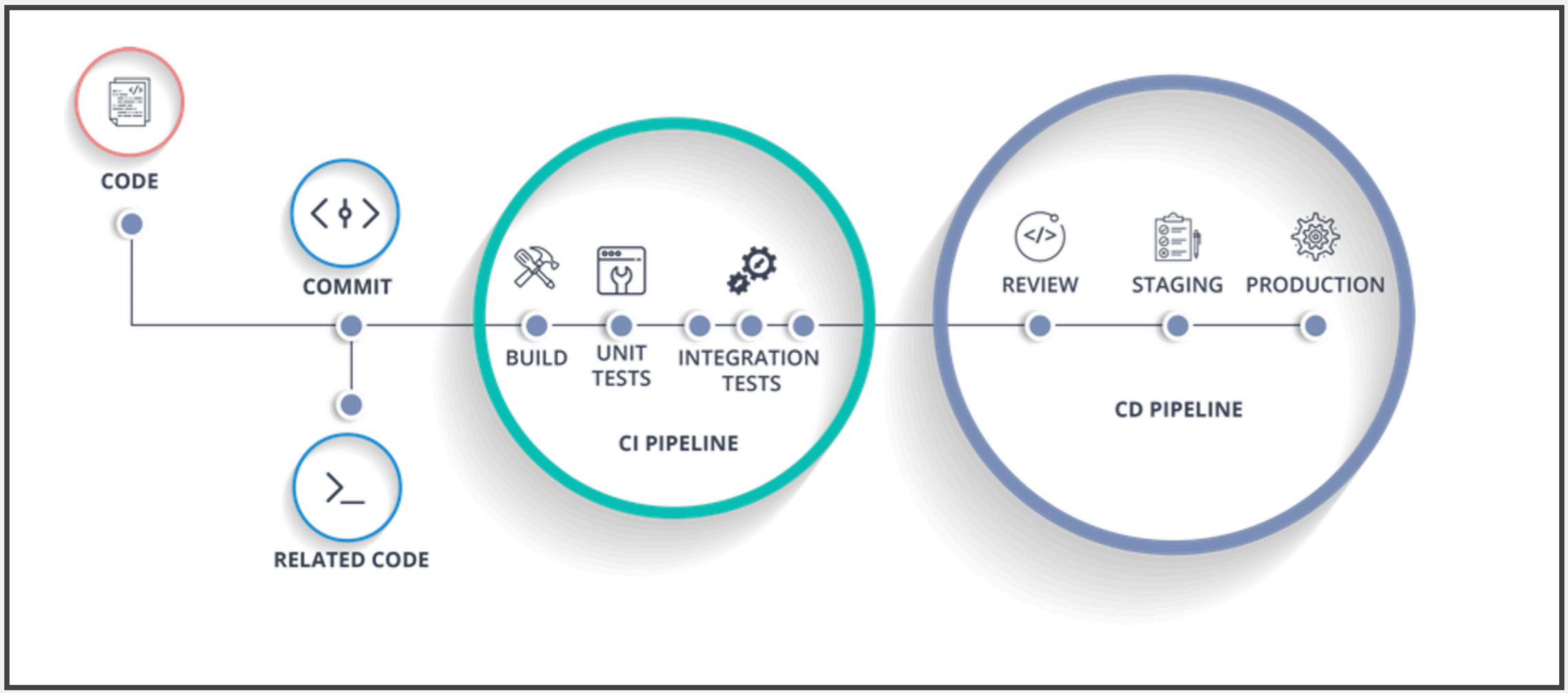
Submit unit tests for your changes. Go has a great test framework built in; use it! Take a look at existing te inspiration. Run the full test suite on your branch before submitting a pull request.

Regression Testing

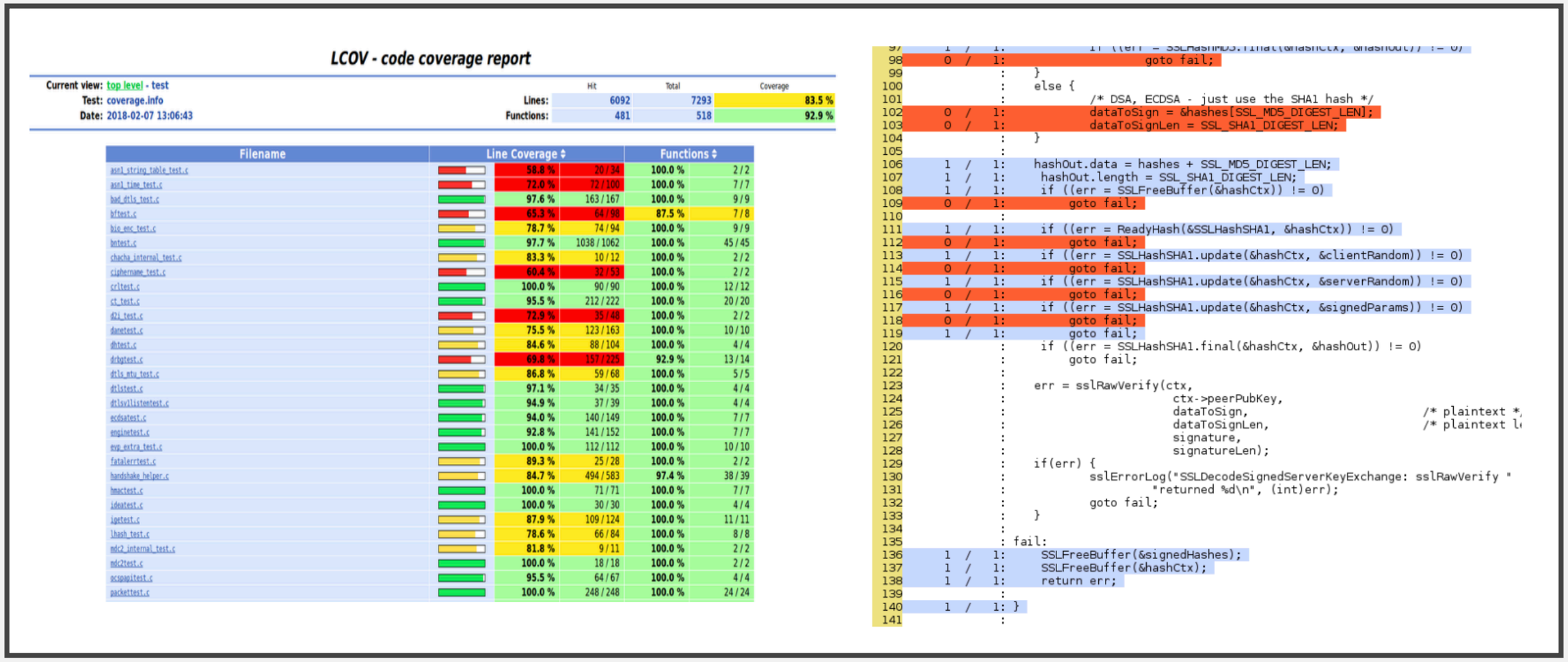


- Usual model:
  - Introduce regression tests for bug fixes, etc.
  - Compare results as code evolves
    - Code1 + TestSet -> TestResults1
    - Code2 + TestSet -> TestResults2
  - As code evolves, compare **TestResults1** with **TestResults2**, etc.
- Benefits:
- Ensure bug fixes remain in place and bugs do not reappear.
- Reduces reliance on specifications, as <**TestSet,TestResults1**> acts as one.

Continuous Integration & Deployment



Code Coverage



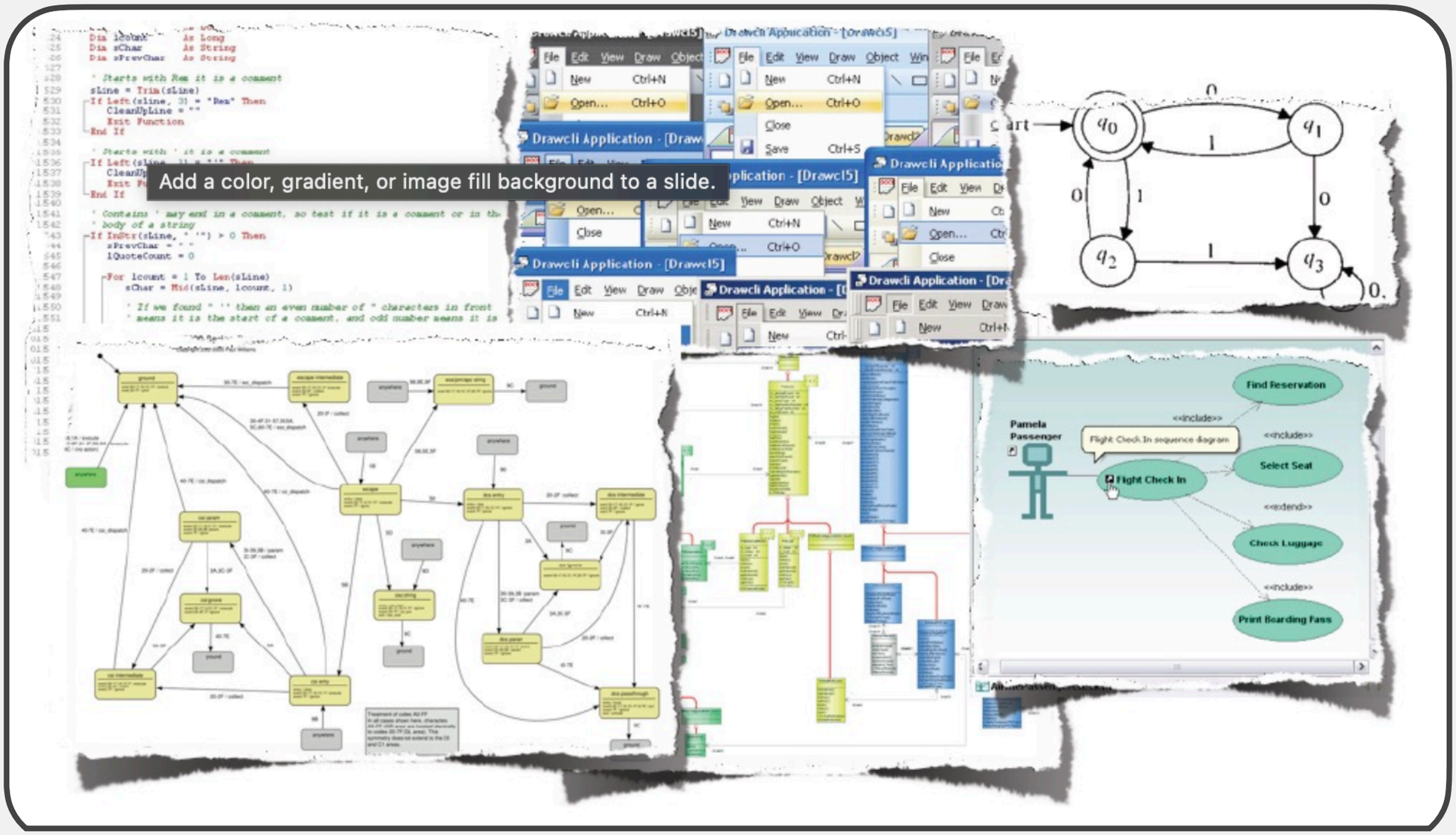
LCOV - code coverage report

		Hit	Total	Coverage
Current view: top level - test	Lines:	6092	7293	83.5 %
Test: coverage.info	Functions:	481	518	92.9 %
Date: 2018-02-07 13:06:43				

Filename	Line Coverage		Functions	
asn1_string_table_test.c	58.8 %	20 / 34	100.0 %	2 / 2
asn1_time_test.c	72.0 %	72 / 100	100.0 %	7 / 7
bad_dtls_test.c	97.6 %	163 / 167	100.0 %	9 / 9
bftest.c	65.3 %	64 / 98	87.5 %	7 / 8
bio_enc_test.c	78.7 %	74 / 94	100.0 %	9 / 9
bntest.c	97.7 %	1038 / 1062	100.0 %	45 / 45
chacha_internal_test.c	83.3 %	10 / 12	100.0 %	2 / 2
ciphername_test.c	60.4 %	32 / 53	100.0 %	2 / 2
crltest.c	100.0 %	90 / 90	100.0 %	12 / 12
ct_test.c	95.5 %	212 / 222	100.0 %	20 / 20
d2i_test.c	72.9 %	35 / 48	100.0 %	2 / 2
danetest.c	75.5 %	123 / 163	100.0 %	10 / 10
dhtest.c	84.6 %	88 / 104	100.0 %	4 / 4
drbgtest.c	69.8 %	157 / 225	92.9 %	13 / 14
dtls_mtu_test.c	86.8 %	59 / 68	100.0 %	5 / 5
dtlstest.c	97.1 %	34 / 35	100.0 %	4 / 4
dtlsvllistentest.c	94.9 %	37 / 39	100.0 %	4 / 4
ecdsatest.c	94.0 %	140 / 149	100.0 %	7 / 7
enginetest.c	92.8 %	141 / 152	100.0 %	7 / 7
evp_extra_test.c	100.0 %	112 / 112	100.0 %	10 / 10
fatalerrtest.c	89.3 %	25 / 28	100.0 %	2 / 2
handshake_helper.c	84.7 %	494 / 583	97.4 %	38 / 39
hmactest.c	100.0 %	71 / 71	100.0 %	7 / 7
ideatest.c	100.0 %	30 / 30	100.0 %	4 / 4
igetest.c	87.9 %	109 / 124	100.0 %	11 / 11
lhash_test.c	78.6 %	66 / 84	100.0 %	8 / 8
mdc2_internal_test.c	81.8 %	9 / 11	100.0 %	2 / 2
mdc2test.c	100.0 %	18 / 18	100.0 %	2 / 2
ocspapitest.c	95.5 %	64 / 67	100.0 %	4 / 4
packettest.c	100.0 %	248 / 248	100.0 %	24 / 24

```
                1  /   1:                if ((err = SSLHashMD5.final(&hashCtx, &hashOut)) != 0)
      0  /   1:                    goto fail;
                :    }
                :    else {
                :        /* DSA, ECDSA - just use the SHA1 hash */
      0  /   1:        dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
      0  /   1:        dataToSignLen = SSL_SHA1_DIGEST_LEN;
                :    }
                :
      1  /   1:    hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
      1  /   1:     hashOut.length = SSL_SHA1_DIGEST_LEN;
      1  /   1:     if ((err = SSLFreeBuffer(&hashCtx)) != 0)
      0  /   1:         goto fail;
                :
      1  /   1:     if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
      0  /   1:         goto fail;
      1  /   1:     if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
      0  /   1:         goto fail;
      1  /   1:     if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
      0  /   1:         goto fail;
      1  /   1:     if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
      0  /   1:         goto fail;
      1  /   1:         goto fail;
                :     if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
                :         goto fail;
                :
                :    err = sslRawVerify(ctx,
                :                       ctx->peerPubKey,
                :                       dataToSign,                     /* plaintext *,
                :                       dataToSignLen,                  /* plaintext l
                :                       signature,
                :                       signatureLen);
                :    if(err) {
                :            sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                :                    "returned %d\n", (int)err);
                :            goto fail;
                :    }
                :
                : fail:
      1  /   1:     SSLFreeBuffer(&signedHashes);
      1  /   1:     SSLFreeBuffer(&hashCtx);
      1  /   1:     return err;
                :
      1  /   1: }
                :
```

We Can Measure Coverage on Almost Anything



Be Aware of Coverage Chasing



- Recall: issues with metrics and incentives
  - Also: Numbers can be deceptive

- 100% coverage != exhaustively tested
  - "Coverage is not strongly correlated with suite effectiveness"

- Based on empirical study on GitHub projects [Inozemtseva and Holmes, ICSE'14]

- Still, it's a good low bar
  - Code that is not executed has definitely not been tested

Coverage of What?



- Distinguish code being tested and code being executed
- Library code >>>> Application code
  - Can selectively measure coverage
- All application code >>> code being tested
  - Not always easy to do this within an application

Coverage != Outcome



- What's better, tests that always pass or tests that always fail?
- Tests should ideally be falsifiable. Boundary determines
- specification
- Ideally:
  - Correct implementations should pass all tests
  - Buggy code should fail at least one test
  - Intuition behind mutation testing (we'll revisit this next week)
- What if tests have bugs?
  - Pass on buggy code or fail on correct code
- Even worse: flaky tests
  - Pass or fail on the same test case nondeterministically
- What's the worst type of test?

Test Design Principles



- Use public APIs only
- Clearly distinguish inputs, configuration, execution, and oracle
- Be simple; avoid complex control flow such as conditionals and loops
- Tests shouldn’t need to be frequently changed or refactored
  - Definitely not as frequently as the code being tested changes

Anti-Patterns



- Snoopy oracles
  - Relying on implementation state instead of observable behavior
  - E.g. Checking variables or fields instead of return values
- Brittle tests
  - Overfitting to special-case behavior instead of general principle
  - E.g. hard-coding message strings instead of behavior
- Slow tests
  - Self-explanatory(beware of heavy environments, I/O, and sleep())
- Flaky tests
  - Tests that pass or fail nondeterministically
  - Often because of reliance on random inputs, timing (e.g. sleep(1000)), availability of external services (e.g. fetching data over the network in a unit test), or dependency on order of test execution (e.g. previous test sets up global variables in certain way)

Takeaways



- Most tests that you will write will be muuuuuuch more complex than testing a sort function.
- Need to set up environment, create objects whose methods to test, create objects for test data, get all these into an interesting state, test multiple APIs with varying arguments, etc.
- Many tests will require mocks (i.e., faking a resource-intensive component).
- General principles of many of these strategies still apply:
  - Writing tests can be time consuming
  - Determining test adequacy can be hard (if not impossible)
  - Test oracles are not easy
  - Advanced test strategies have trade-offs (high costs with high returns)

Week 4- Software Architecture



Abstracted Views Focus on Conveying Information



- They have a well-defined purpose
- Show only necessary information
- Abstract away unnecessary details
- Use legends/annotations to remove ambiguity
- Multiple views of the same object tell a larger story

Levels of Abstraction



- Requirements
  - high-level “what” needs to be done
- Architecture (High-level design)
  - high-level “how”, mid-level “what”
- OO-Design (Low-level design, e.g. design patterns)
  - mid-level “how”, low-level “what”
- Code
  - low-level “how”

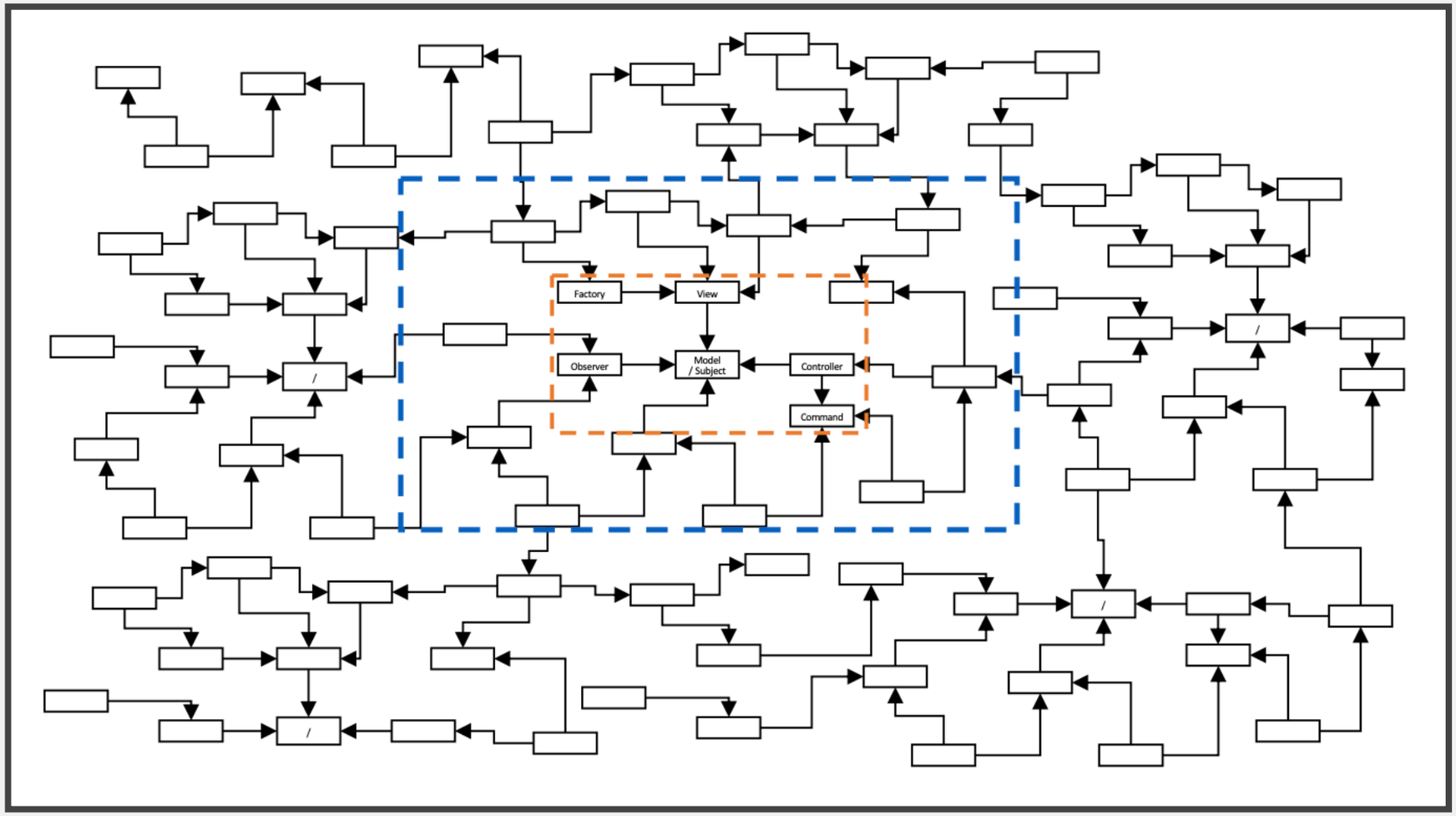
Design vs. Architecture



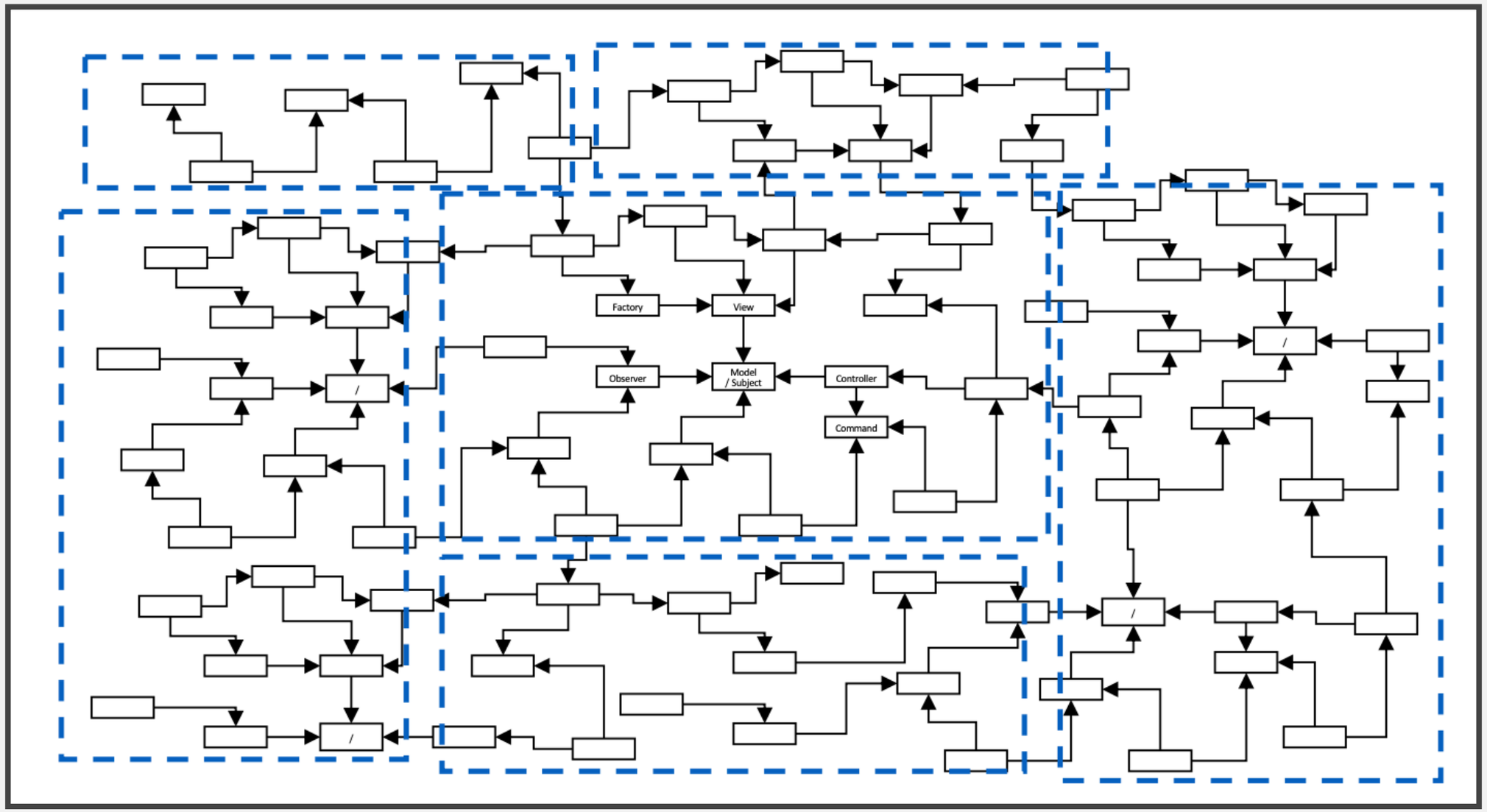
- Design Questions
  - *How do I add a menu item in VSCode?*
  - *How can I make it easy to add menu items in VSCode?*
  - *What lock protects this data?*
  - *How does Google rank pages?*
  - *What encoder should I use for secure communication?*
  - *What is the interface between objects?*

- Architectural Questions
  - *How do I extend VSCode with a plugin?*
  - *What threads exist and how do they coordinate?*
  - *How does Google scale to billions of hits per day?*
  - *Where should I put my firewalls?*
  - *What is the interface between subsystems?*

Design Patterns



Architecture



Why Document Architecture?



- Blueprint for the system
  - Artifact for early analysis
  - Primary carrier of quality attributes
  - Key to post-deployment maintenance and enhancement

- Documentation speaks for the architect, today and 20 years from today
  - As long as the system is built, maintained, and evolved according to its documented architecture

- Support traceability.

Views & Purposes



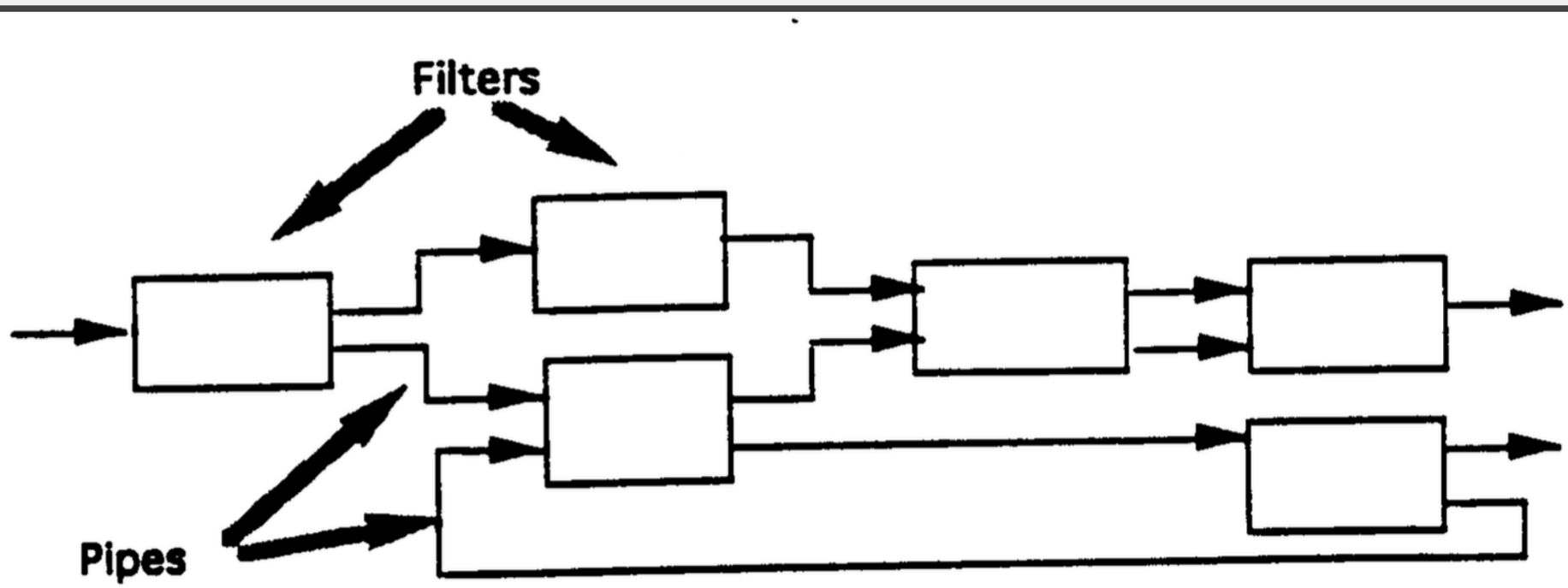
- Every view should align with a purpose
- • Views should only represent information relevant to that purpose
  - Abstract away other details
  - Annotate view to guide understanding where needed
- • Different views are suitable for different reasoning aspects (different quality goals), e.g.,
  - Performance
  - Extensibility
  - Security
  - Scalability
  - ...

Common Views in Documenting Software Architecture



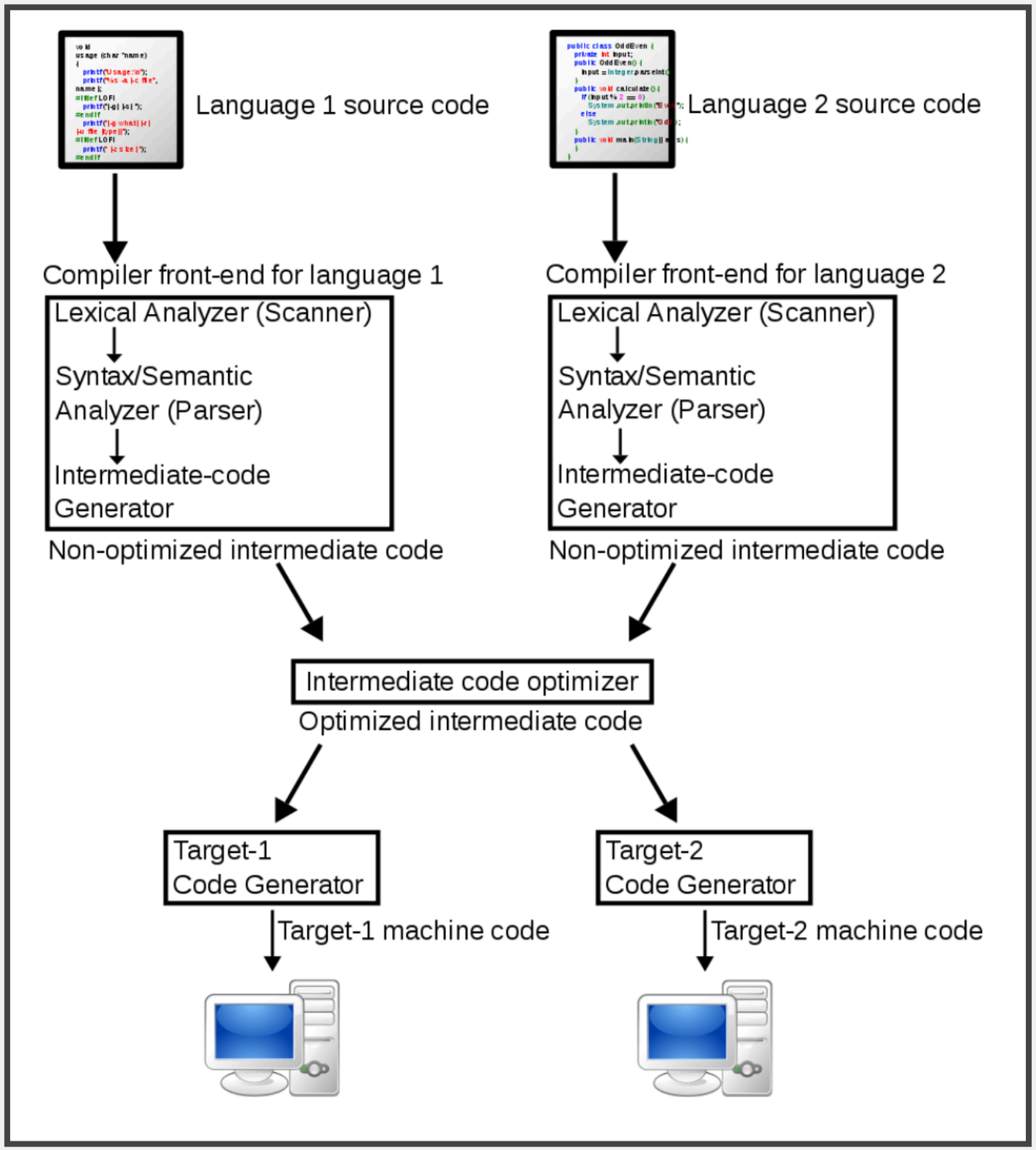
- Static View
  - Modules (subsystems, structures) and their relations (dependencies, ...)
- Dynamic View
  - Components (processes, runnable entities) and connectors (messages, data flow, ...)
- Physical View (Deployment)
  - Hardware structures and their connections

1. Pipes & Filters

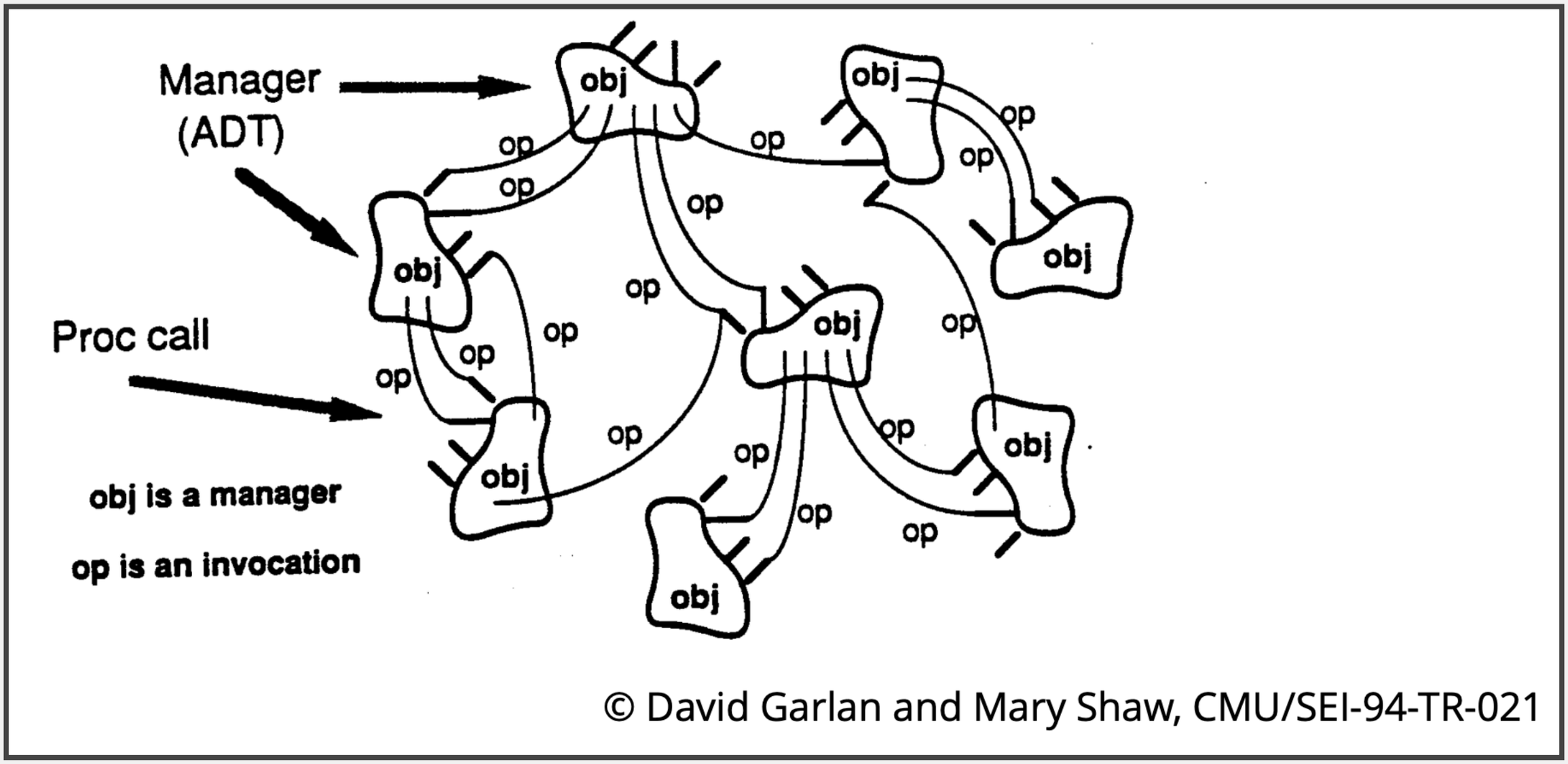


© David Garlan and Mary Shaw, CMU/SEI-94-TR-021

Pipes & Filters Example: Compilers

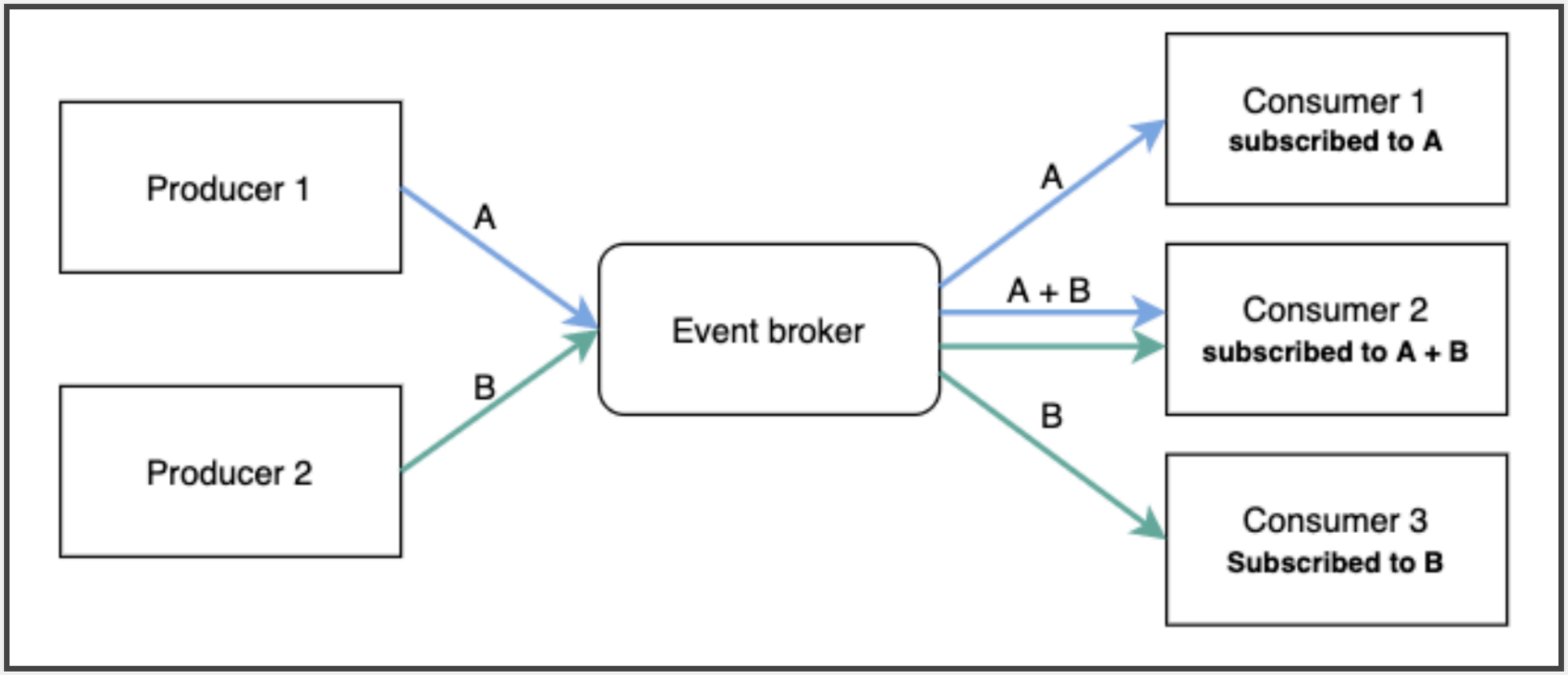


2. Object Oriented Organization



© David Garlan and Mary Shaw, CMU/SEI-94-TR-021

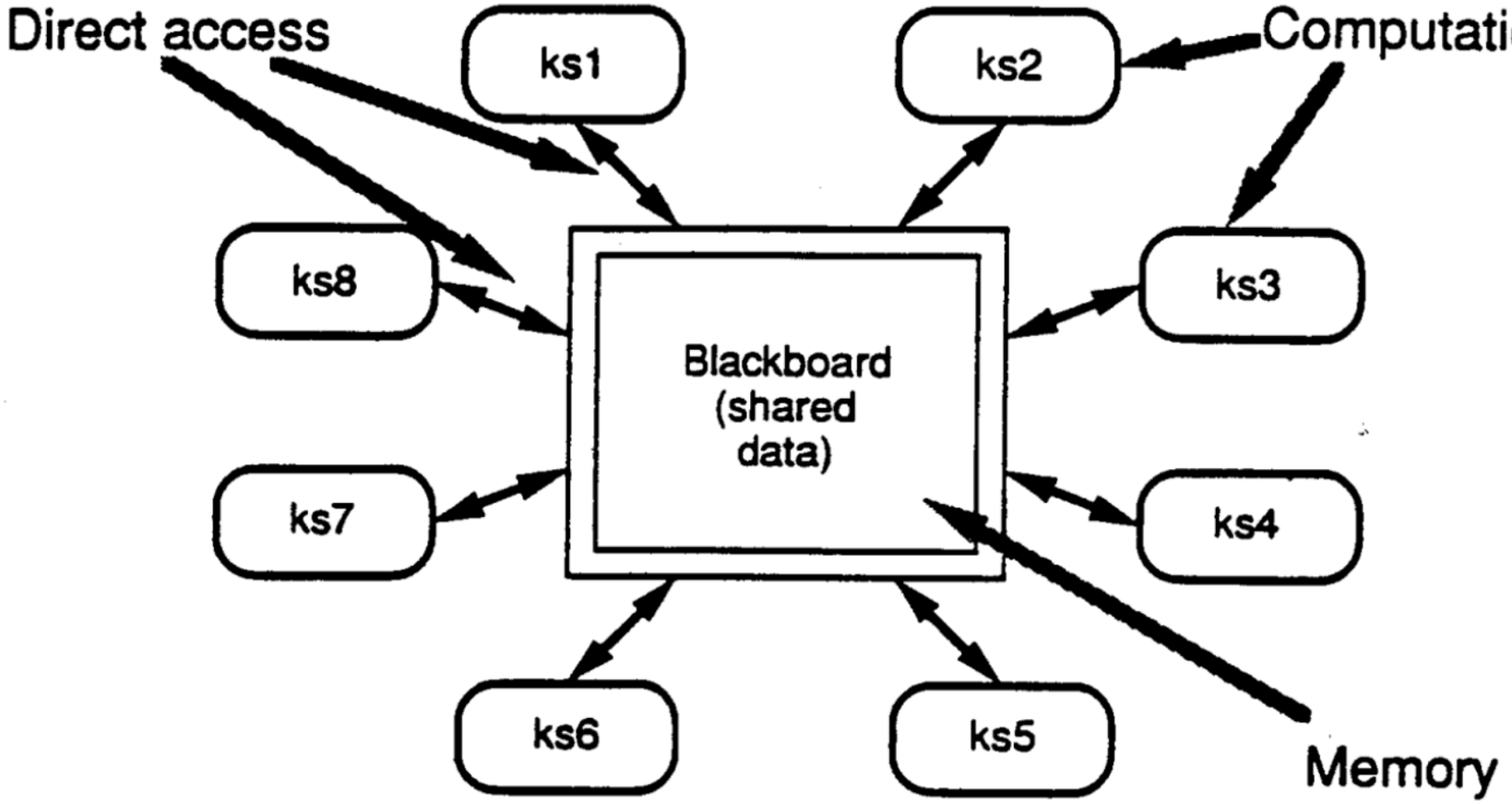
3. Event-Driven Architecture



Example: HTML DOM + Javascript

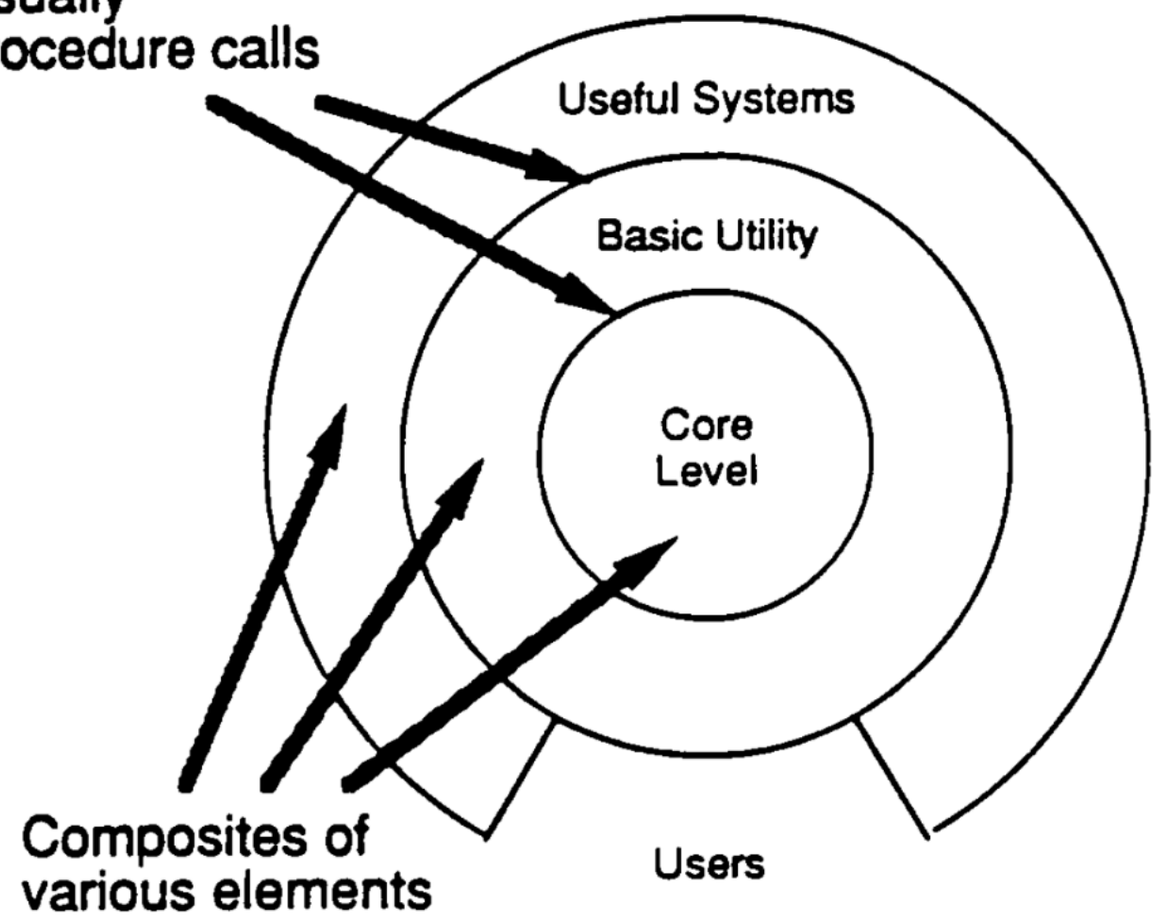


4. Blackboard Architecture



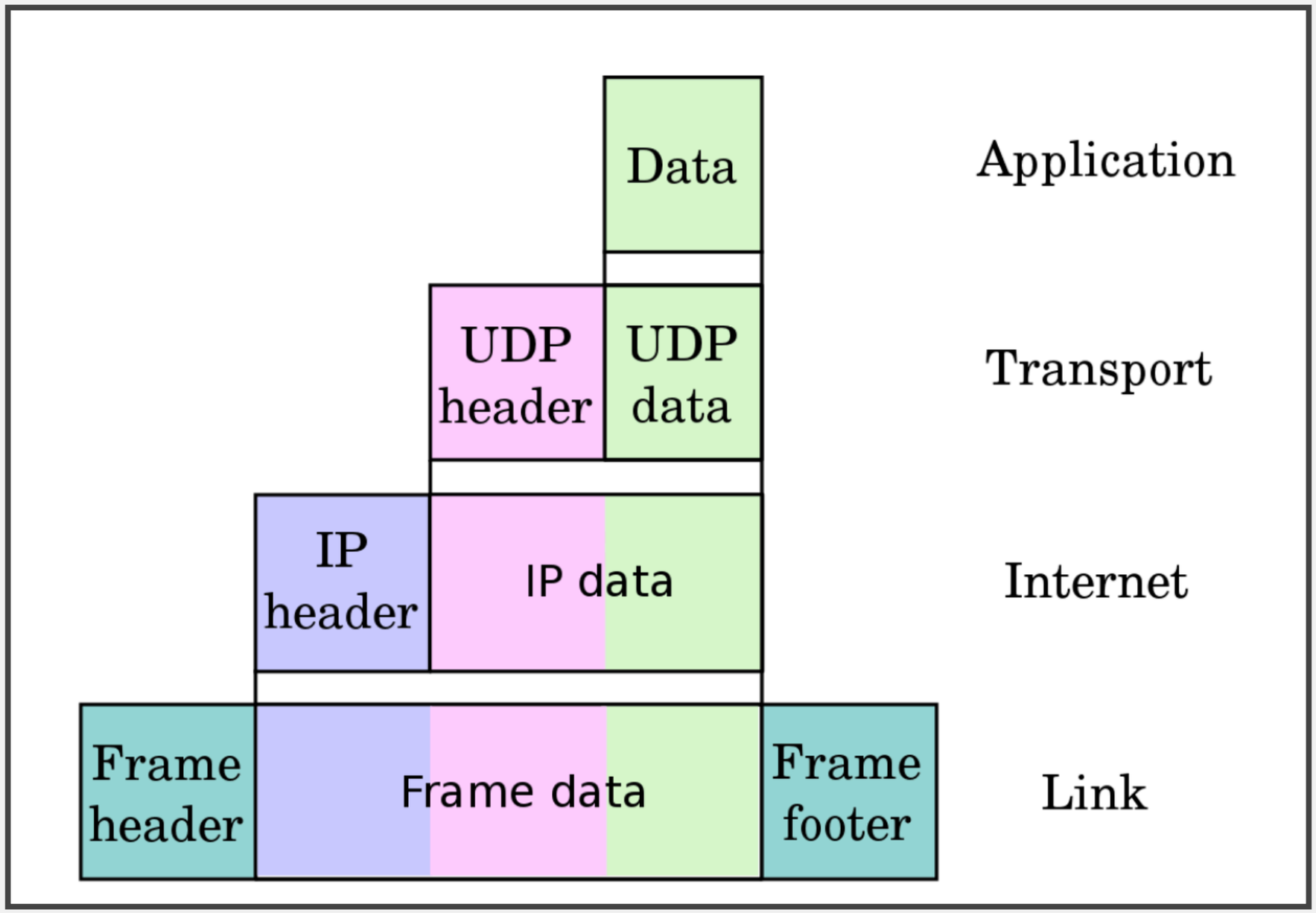
© David Garlan and Mary Shaw, CMU/SEI-94-TR-021

5. Layered Systems



© David Garlan and Mary Shaw, CMU/SEI-94-TR-021

Example Internet Protocol Suite



Guidelines for Selecting a Notation



- Suitable for purpose
- Often visual for compact representation
- Usually boxes and arrows
- UML possible (semi-formal), but possibly constraining
  - Note the different abstraction level – Subsystems or processes, not classes or objects
- Formal notations available
- Decompose diagrams hierarchically and in views
- Always include a legend
- Define precisely what the boxes mean
- Define precisely what the lines mean
- Do not try to do too much in one diagram
  - Each view of architecture should fit on a page
  - Use hierarchy

Week 5 - Static and Dynamic Analysis



What Static Analysis Can & Cannot Do



- Type-checking is well established
  - Set of data types taken by variables at any point
  - Can be used to prevent type errors (e.g. Java) or warn about potential type errors (e.g. Python)

- Checking for problematic patterns in syntax is easy and fast
  - Is there a comparison of two Java strings using `==`?
  - Is there an array access `a[i]` without an enclosing bounds check for `i`?

- Reasoning about termination is impossible in general
  - Halting problem
- Reasoning about exact values is hard, but conservative analysis via abstraction is possible
  - Is the bounds check before `a[i]` guaranteeing that `I` is within bounds?
  - Can the divisor ever take on a zero value?
  - Could the result of a function call be `42`?
  - Will this multi-threaded program give me a deterministic result?
  - Be prepared for “MAYBE”

- Verifying some advanced properties is possible but expensive
  - CI-based static analysis usually over-approximates conservatively

Bad News: Rice's Theorem



- Every static analysis is necessarily incomplete, unsound, undecidable, or a combination thereof

- *"Any nontrivial property about the language recognized by a Turing machine is undecidable."*
  - Henry Gordon Rice, 1953

Static Analysis is Well-Suited to Detecting Certain Defects



- **Security:** Buffer overruns, improperly validated input...
- **Memory safety:** Null dereference, uninitialized data...
- **Resource leaks:** Memory, OS resources...
- **API Protocols:** Device drivers; real time libraries; GUI frameworks
- **Exceptions:** Arithmetic/library/user-defined
- **Encapsulation:**
  - Accessing internal data, calling private functions...
- **Data races:**
  - Two threads access the same data without synchronization

Static Analysis Tools: Broad Classification



- Linters
  - Shallow syntax analysis for enforcing code styles and formatting
- Pattern-based bug detectors
  - Simple syntax or API-based rules for identifying common programming mistakes
- Type-annotation validators
  - Check conformance to user-defined types
  - Types can be complex (e.g., “Nullable”)
- Data-flow analysis / Abstract interpretation)
  - Deep program analysis to find complex error conditions (e.g., ” can array index be out of bounds?”)

Static Analysis Tools: Applications



- Find bugs
- Refactor code
- Keep your code stylish!
- Identify code smells
- Measure quality
- Find usability and accessibility issues
- Identify bottlenecks and improve performance

Dynamic Analysis Reasons about Program Executions



- Tells you properties of the program that were definitely observed
  - Code coverage
  - Performance profiling
  - Type profiling
  - Testing
- In practice, implemented by program instrumentation
  - Think “Automated logging”
  - Slows down execution speed by a small amount

Static Analysis vs. Dynamic Analysis



- Requires only source code
- Conservatively reasons about all possible
- Reported warnings may contain false positives
- Can report all warnings of a particular class of problems
- Advanced techniques like verification can prove certain complex properties, but rarely run in CI due to cost

- Requires successful build + test inputs
- Observes individual executions
- Reported problems are real, as observed by a witness input
- Can only report problems that are seen. Highly dependent on test inputs. Subject to false negatives
- Advanced techniques like symbolic execution can prove certain complex properties, but rarely run in CI due to cost

Tools for Static Analysis













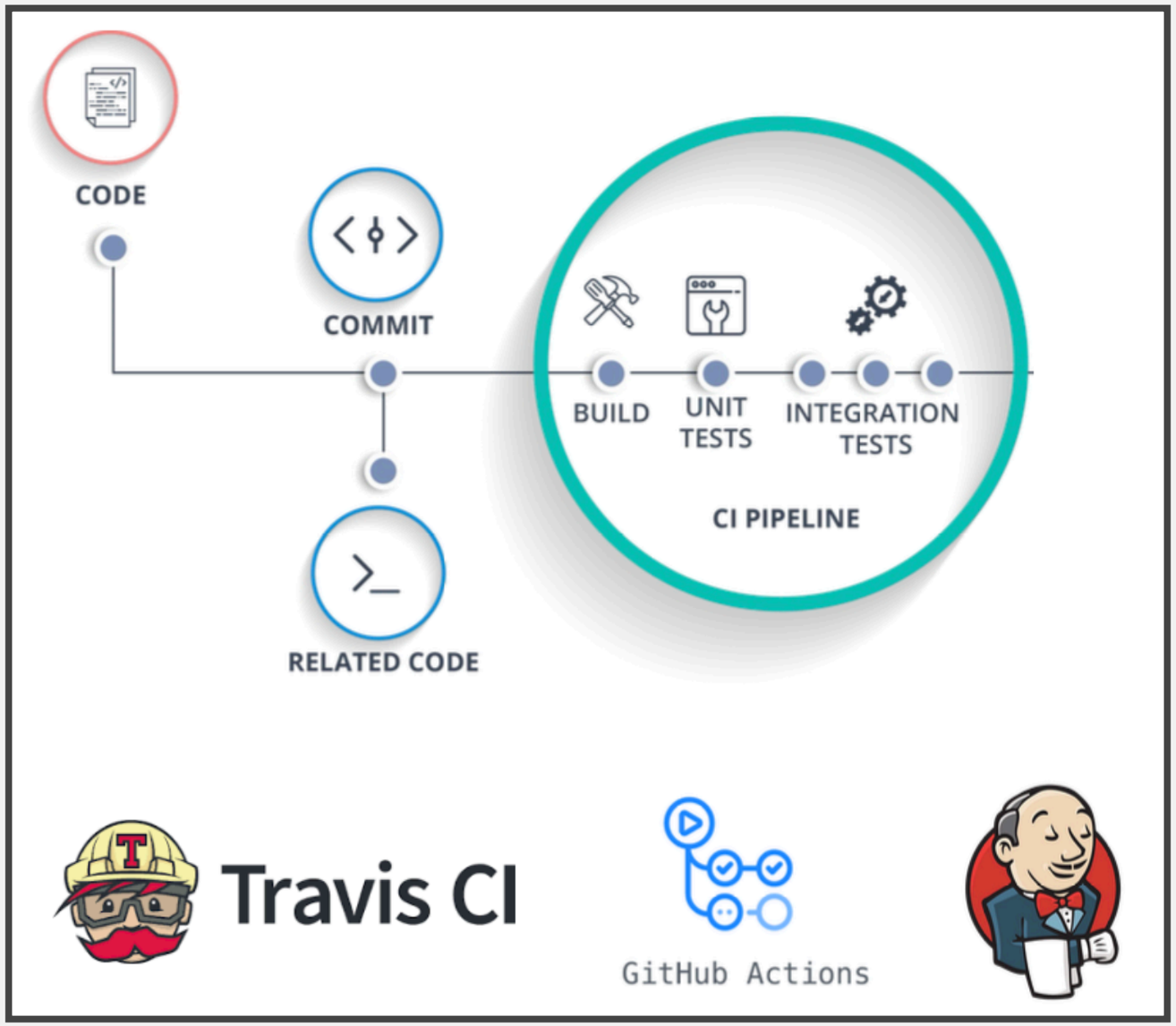








Static Analysis is a Key Part of CI



What Makes a Good Static Analysis Tool?



- Static analysis should be fast
  - Don’t hold up development velocity
  - This becomes more important as code scales
- Static analysis should report few false positives
  - Otherwise developers will start to ignore warnings and alerts, and quality will decline
- Static analysis should be continuous
  - Should be part of your continuous integration pipeline
  - Diff-based analysis is even better -- don’t analyse the entire codebase; just the changes
- Static analysis should be informative
  - Messages that help the developer to quickly locate and address the issue
  - Ideally, it should suggest or automatically apply fixes

(1) Linters



- Cheap, fast, and lightweight static source analysis

Linters Use Very "Shallow" Static Analysis



- Ensure proper indentation
- Naming convention
- Line sizes
- Class nesting
- Documenting public functions
- Parenthesis around expressions
- What else?

Use Linters to Improve Maintainability

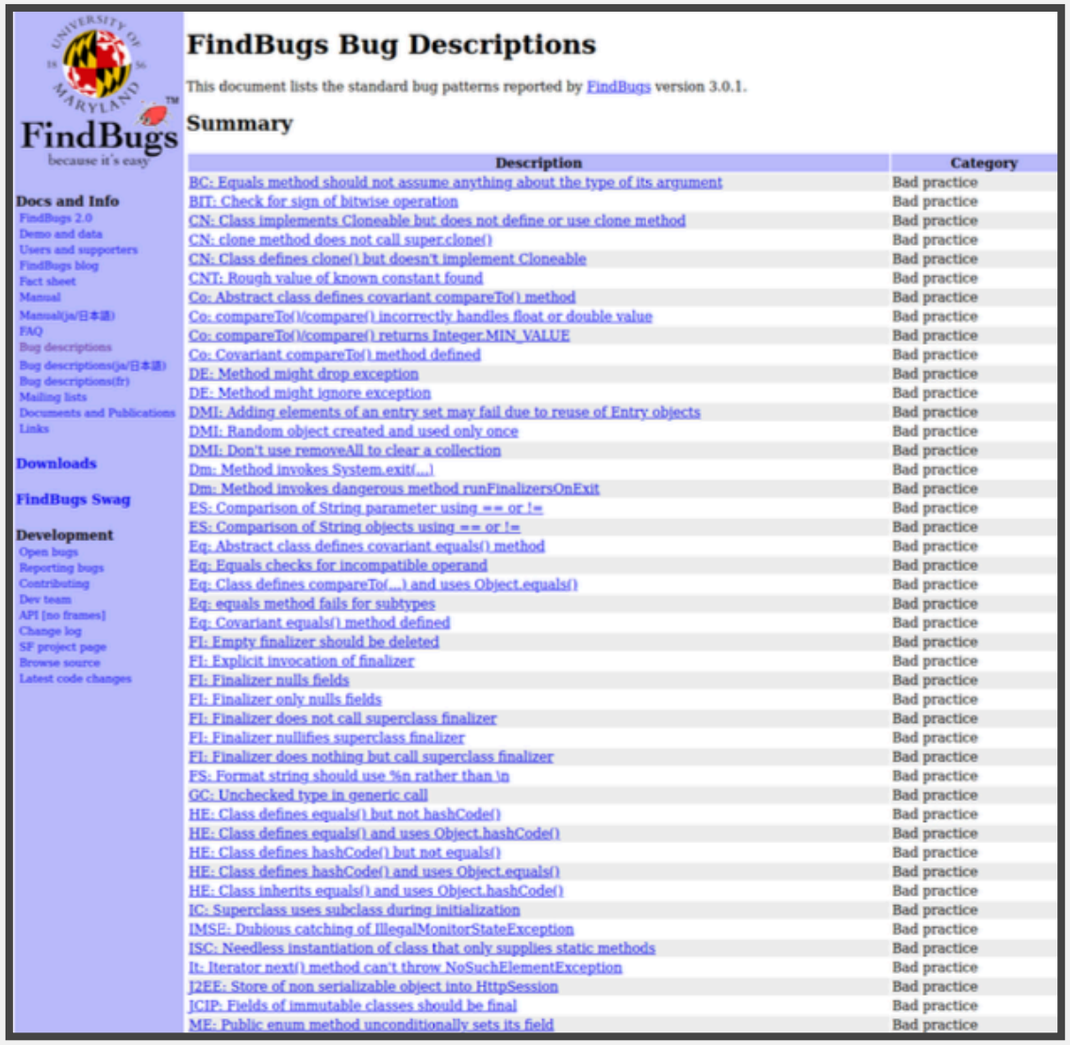


- Why? We spend more time reading code than writing it.
  - Various estimates of the exact %, some as high as 80%
- Code is ownership is usually shared
- The original owner of some code may move on
- Code conventions make it easier for other developers to quickly understand your code

(2) - Pattern-based Static Analysis Tools



- Bad Practice
- Correctness
- Performance
- Internationalization
- Malicious Code
- Multithreaded Correctness
- Security
- Dodgy Code



FindBugs
because it's easy

Docs and Info
FindBugs 2.0
Demo and data
Users and supporters
FindBugs blog
Fact sheet
Manual
Manual(ja/日本語)
FAQ
Bug descriptions
Bug descriptions(ja/日本語)
Bug descriptions(fr)
Mailing lists
Documents and Publications
Links

Downloads

FindBugs Swag

Development
Open bugs
Reporting bugs
Contributing
Dev team
API [no frames]
Change log
SF project page
Browse source
Latest code changes

FindBugs Bug Descriptions

This document lists the standard bug patterns reported by FindBugs version 3.0.1.

Summary

Description	Category
BC: Equals method should not assume anything about the type of its argument	Bad practice
BIT: Check for sign of bitwise operation	Bad practice
CN: Class implements Cloneable but does not define or use clone method	Bad practice
CN: clone method does not call super.clone()	Bad practice
CN: Class defines clone() but doesn't implement Cloneable	Bad practice
CNT: Rough value of known constant found	Bad practice
Co: Abstract class defines covariant compareTo() method	Bad practice
Co: compareTo()/compare() incorrectly handles float or double value	Bad practice
Co: compareTo()/compare() returns Integer.MIN_VALUE	Bad practice
Co: Covariant compareTo() method defined	Bad practice
DE: Method might drop exception	Bad practice
DE: Method might ignore exception	Bad practice
DMI: Adding elements of an entry set may fail due to reuse of Entry objects	Bad practice
DMI: Random object created and used only once	Bad practice
DMI: Don't use removeAll to clear a collection	Bad practice
Dm: Method invokes System.exit(...)	Bad practice
Dm: Method invokes dangerous method runFinalizersOnExit	Bad practice
ES: Comparison of String parameter using == or !=	Bad practice
ES: Comparison of String objects using == or !=	Bad practice
Eq: Abstract class defines covariant equals() method	Bad practice
Eq: Equals checks for incompatible operand	Bad practice
Eq: Class defines compareTo(...) and uses Object.equals()	Bad practice
Eq: equals method fails for subtypes	Bad practice
Eq: Covariant equals() method defined	Bad practice
FI: Empty finalizer should be deleted	Bad practice
FI: Explicit invocation of finalizer	Bad practice
FI: Finalizer nulls fields	Bad practice
FI: Finalizer only nulls fields	Bad practice
FI: Finalizer does not call superclass finalizer	Bad practice
FI: Finalizer nullifies superclass finalizer	Bad practice
FI: Finalizer does nothing but call superclass finalizer	Bad practice
FS: Format string should use %n rather than \n	Bad practice
GC: Unchecked type in generic call	Bad practice
HE: Class defines equals() but not hashCode()	Bad practice
HE: Class defines equals() and uses Object.hashCode()	Bad practice
HE: Class defines hashCode() but not equals()	Bad practice
HE: Class defines hashCode() and uses Object.equals()	Bad practice
HE: Class inherits equals() and uses Object.hashCode()	Bad practice
IC: Superclass uses subclass during initialization	Bad practice
IMSE: Dubious catching of IllegalMonitorStateException	Bad practice
ISC: Needless instantiation of class that only supplies static methods	Bad practice
It: Iterator next() method can't throw NoSuchElementException	Bad practice
J2EE: Store of non serializable object into HttpSession	Bad practice
JCIP: Fields of immutable classes should be final	Bad practice
ME: Public enum method unconditionally sets its field	Bad practice

Challenges



- The analysis must produce zero false positives
  - Otherwise developers won't be able to build the code!

- The analysis needs to be really fast
  - Ideally < 100 ms
  - If it takes longer, developers will become irritated and lose productivity

- You can't just "turn on" a particular check
  - Every instance where that check fails will prevent existing code from
  - There could be thousands of violations for a single check across large codebases

(3) -Use Type Annotations to Detect Common Errors



- Uses a conservative analysis to prove the absence of certain defects
  - Null pointer errors, uninitialized fields, certain liveness issues, information leaks, SQL injections, bad regular expressions, incorrect physical units, bad format strings, ...
    - C.f. SpotBugs which makes no safety guarantees
    - Assuming that code is annotated and those annotations are correct
- Uses annotations to enhance type system
- Example: Java Checker Framework or MyPy



(3) -Use Type Annotations to Detect Common Errors



- Uses a conservative analysis to prove the absence of certain defects
  - Null pointer errors, uninitialized fields, certain liveness issues, information leaks, SQL injections, bad regular expressions, incorrect physical units, bad format strings, ...
    - C.f. SpotBugs which makes no safety guarantees
    - Assuming that code is annotated and those annotations are correct
- Uses annotations to enhance type system
- Example: Java Checker Framework or MyPy



Taint Analysis



- Tracks flow of sensitive information through the program
- Tainted inputs come from arbitrary, possibly malicious sources
  - User inputs, unvalidated data
- Using tainted inputs may have dangerous consequences
  - Program crash, data corruption, leak private data, etc.
- We need to check that inputs are sanitized before reaching sensitive locations

Units Checker Identifies Physical Unit Inconsistencies



- Guarantees that operations are performed on the same kinds and units
- Kinds of annotations
  - @Acceleration, @Angle, @Area, @Current, @Length, @Luminance, @Mass, @Speed, @Substance, @Temperature, @Time
- SI unit annotation
  - @m, @km, @mm, @kg, @mPERs, @mPERs2, @radians, @degrees, @A, ...

Checker Frameworks: Limitations



- Can only analyze code that is annotated
  - Requires that dependent libraries are also annotated
  - Can be tricky, but not impossible, to retrofit annotations into existing codebases

- Only considers the signature and annotations of methods
  - Doesn’t look at the implementation of methods that are being called

- Dynamically generated code
  - Spring Framework

- - Can produce false positives!
  - Byproduct of necessary approximations

Infer: What if we didn't want Annotations



- Focused on memory safety bugs
  - Null pointer dereferences, memory leaks, resource leaks, ...
- Compositional interprocedural reasoning
  - Based on separation logic and bi-abduction
- Scalable and fast
  - Can run incremental analysis on changed code
- Does not require annotations
- Supports multiple languages
  - Java, C, C++, Objective-C
  - Programs are compiled to an intermediate representation



Summary



- Linters are cheap, fast, but imprecise analysis tools
  - Can be used for purposes other than bug detection (e.g., style)
- Conservative analyzers can demonstrate the absence of particular defects
  - At the cost of false positives due to necessary approximations
  - Inevitable trade-off between false positives and false negatives
- The best QA strategy involves multiple analysis and testing techniques
  - The exact set of tools and techniques depends on context



From Autocomplete to Agents

Autocomplete

Suggests the next line as you type.

a line

Chat

Answers a question or writes code on request.

an answer

Agent

Plans, calls tools, checks results, and iterates.

a whole task



you the AI

Your role shifts from **writing** code to **directing and reviewing** it.

Adapted from O. Chaparro (William & Mary)

How LLMs Generate Code

CONTEXT WINDOW: EVERY TOKEN SO FAR

`def` `add` `(` `a` `,` `b` `)` `:`

0

new tokens generated

Model

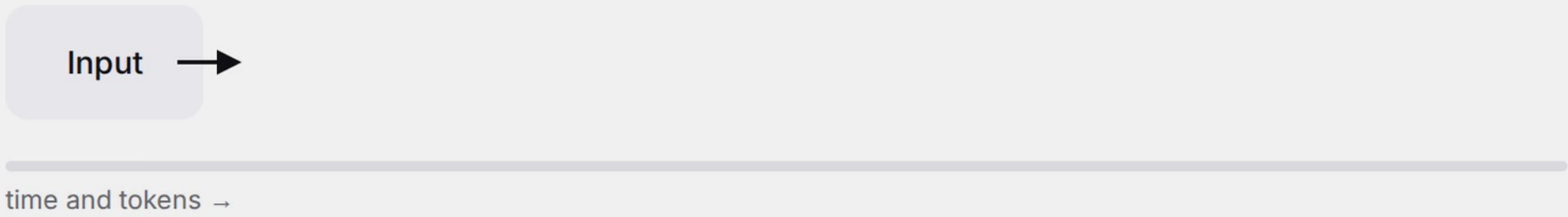
MOST LIKELY NEXT TOKENS

Key implication: the output can only be as good as what is in the context window.

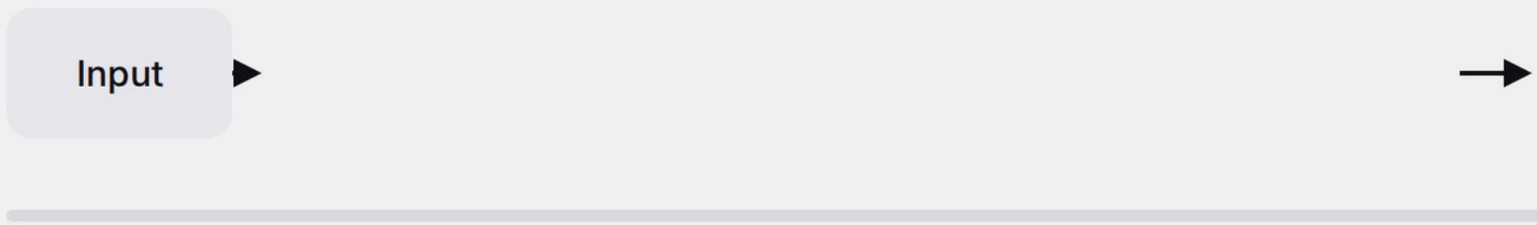
Adapted from O. Chaparro (William & Mary)

Reasoning Models

Standard LLM
answers immediately



Reasoning model
"thinks" before answering



Slower

More tokens, more cost

Better on multi-step problems

Adapted from O. Chaparro (William & Mary)

What Makes an LLM an Agent

LLM
predicts text

+

Loop
observe, act again

+

Tools
act on the world

=

Agent
iterates on a task

TYPICAL TOOLS

read and edit files | terminal | run tests | search | APIs (MCP) | subagents

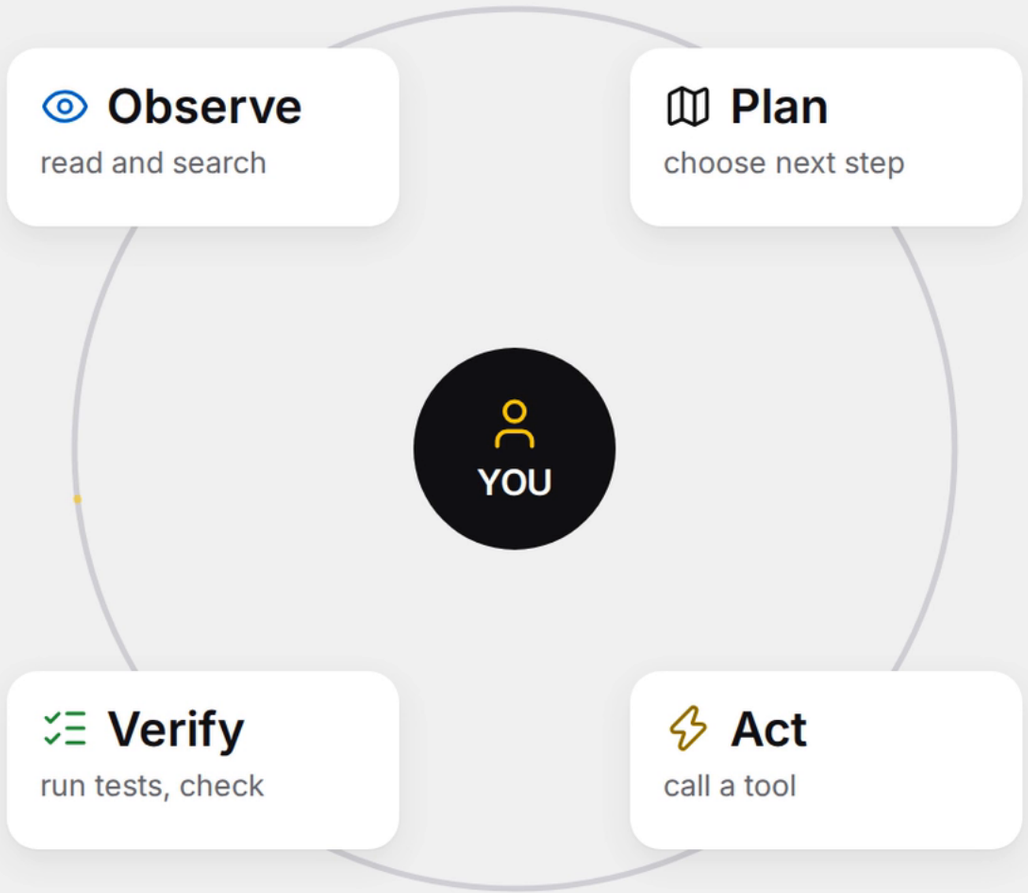
LLM alone
Answers once and forgets. Cannot run your code or check its own work.

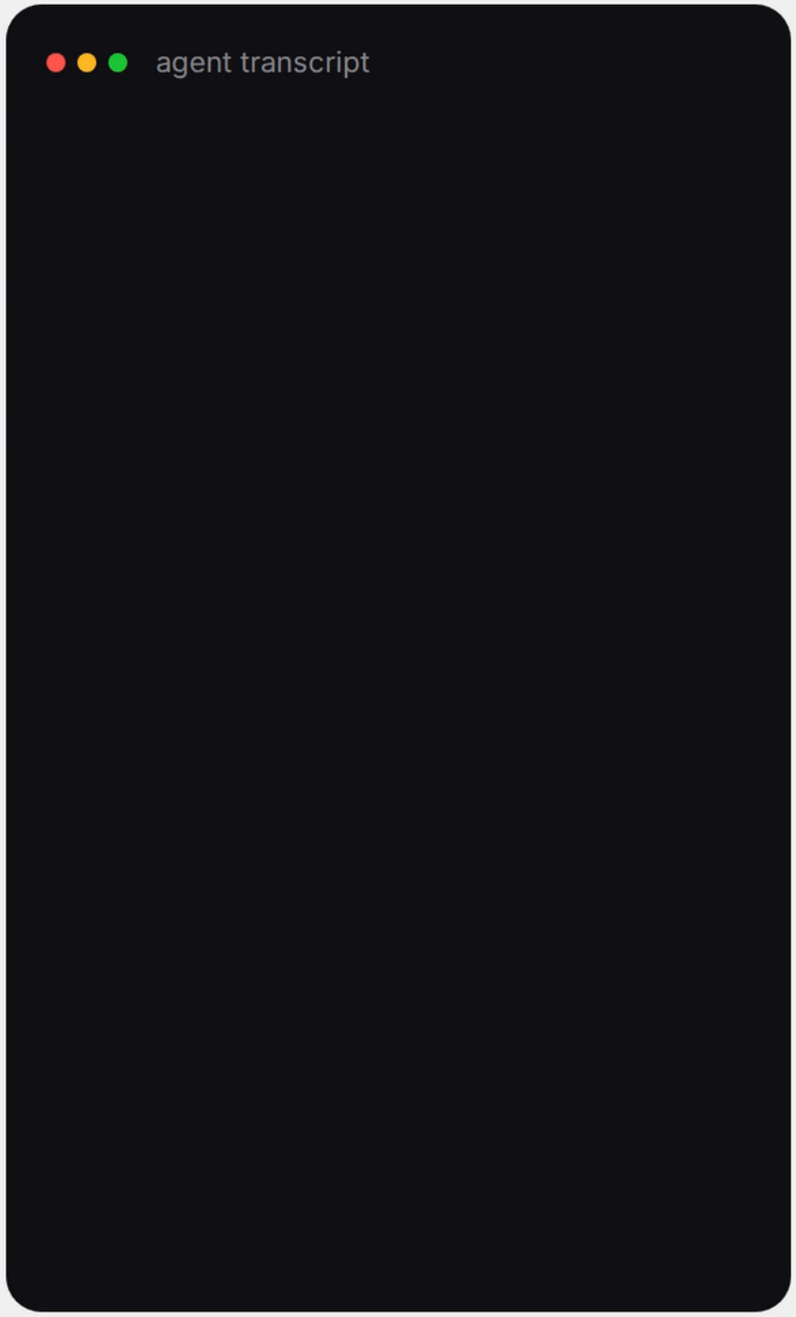
Agent
Reads files, runs commands, sees the results, and tries again.

Adapted from O. Chaparro (William & Mary)

The Agent Loop

Task: fix the failing PNG-export test





Adapted from O. Chaparro (William & Mary)

The Context Window



Adapted from O. Chaparro (William & Mary)

Who Fills the Context Window

CONTEXT WINDOW

System prompt
instructions, rules

Tool definitions
what the agent can call

Instructions file
CLAUDE.md / AGENTS.md

Files you point at
@src/export.ts, specs

Tool results
file contents, test output, logs

Your request
“Fix the export bug…”

You fill

instructions file, pointers, your request

The harness fills

system prompt, tool definitions, results

A shared job.

When the window fills, the harness decides what to trim. Anything that must survive belongs in a file.

Adapted from O. Chaparro (William & Mary)

Memory Across Sessions

SESSION 1

IN-CONTEXT MEMORY

SESSION 2

IN-CONTEXT MEMORY

EXTERNAL MEMORY: FILES ON DISK



CLAUDE.md

Harness Architecture

External services
MCP servers · files · APIs · databases

Harness
prompt · tools · hooks · permissions



LLM

the only part that “thinks”

- System prompt and instructions file
- Tools: files, shell, search, APIs
- Hooks: scripts on lifecycle events
- Permission tiers
- Subagent configurations
- MCP: connects external services

“A harness is largely a delivery mechanism for good context engineering.”

Adapted from O. Chaparro (William & Mary)

What You Configure in a Harness

Build from scratch
only if the agent is your product

You adapt it
instructions file · permissions · MCP servers · subagents · what you point it at



your job

Ships with the tool
the loop · tools · compaction · permission system

You are not building a harness. You are specializing a general one for your project.

Adapted from O. Chaparro (William & Mary)

Coding vs. General Harnesses

	GENERAL HARNESS	CODING HARNESS
Tools	web search, sandbox	read and edit files · grep · shell · git · tests
Context	everything in the window	search the repo, read only what matters
Guidance	generic	read before editing · match conventions · small diffs
Feedback	no ground truth	tests, types, and builds give automatic feedback
Safety	chat cannot break anything	approval gates · branches · sandboxing

Same model, same loop: the difference is tools, context policy, and verification.

Adapted from O. Chaparro (William & Mary)

Interaction Modes

MORE AUTONOMY

Inline

Suggests as you type

YOU REVIEW
every suggestion

Chat

You ask; it generates

YOU REVIEW
every answer

Agentic

Plans and runs many steps

YOU REVIEW
at checkpoints

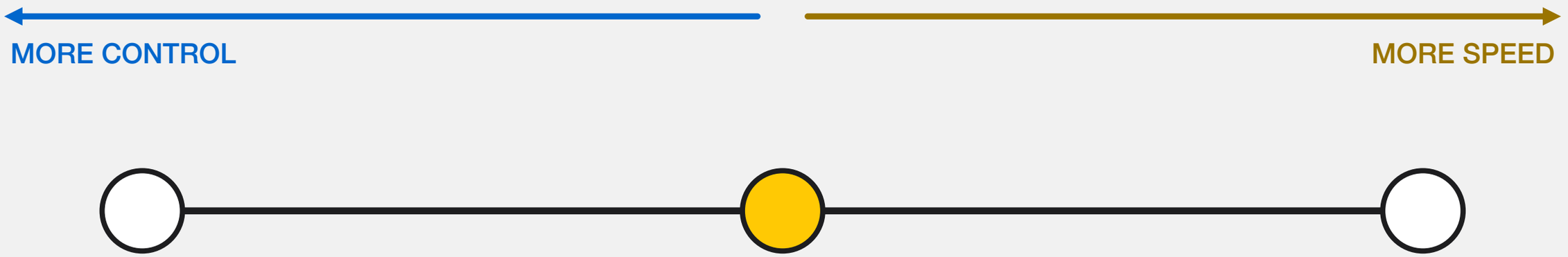
Background

Runs without you

YOU REVIEW
the final result

Adapted from O. Chaparro (William & Mary)

Autonomy Levels



Supervised

Agent proposes each action; you approve before it runs.

Semi-autonomous

Runs until a decision point: a question, a risky file.

Autonomous

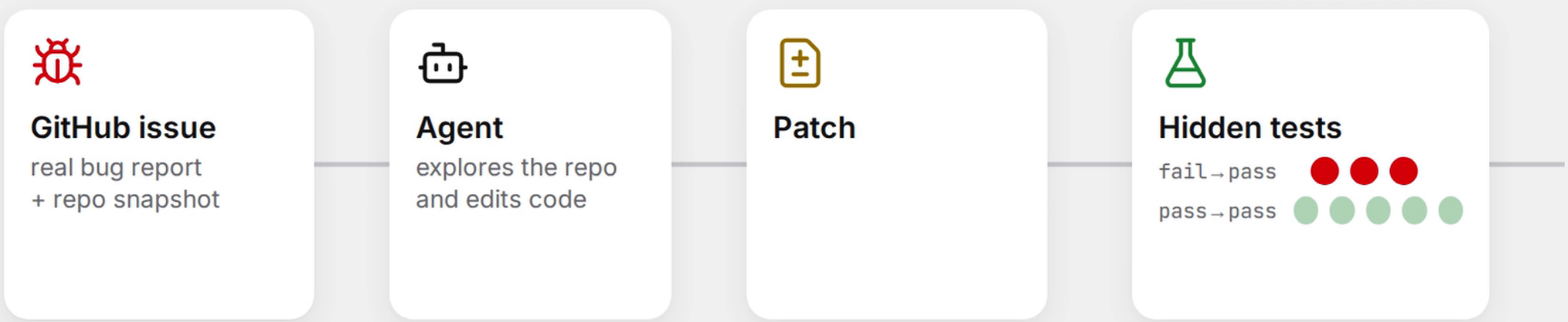
Runs until the task is done. You review the result.

THE RIGHT LEVEL DEPENDS ON

- Task risk
- Familiarity with the code
- Trust built on this kind of task

Adapted from O. Chaparro (William & Mary)

Measuring Agents with SWE-bench



2,294
real issues from 12 Python repos

500
human-validated tasks in "Verified"

1 bit
per task: do the tests pass?

Limits of SWE-bench Scores

7.8%

of “resolved” patches fail the full developer test suite

Wang et al., 2025

29.6%

behave differently from the developer’s real fix

Wang et al., 2025

59.4%

of audited Verified problems had flawed tests

OpenAI, Feb 2026

Contamination

Frontier models had seen some problems and solutions in training. Higher scores partly reflect memory.

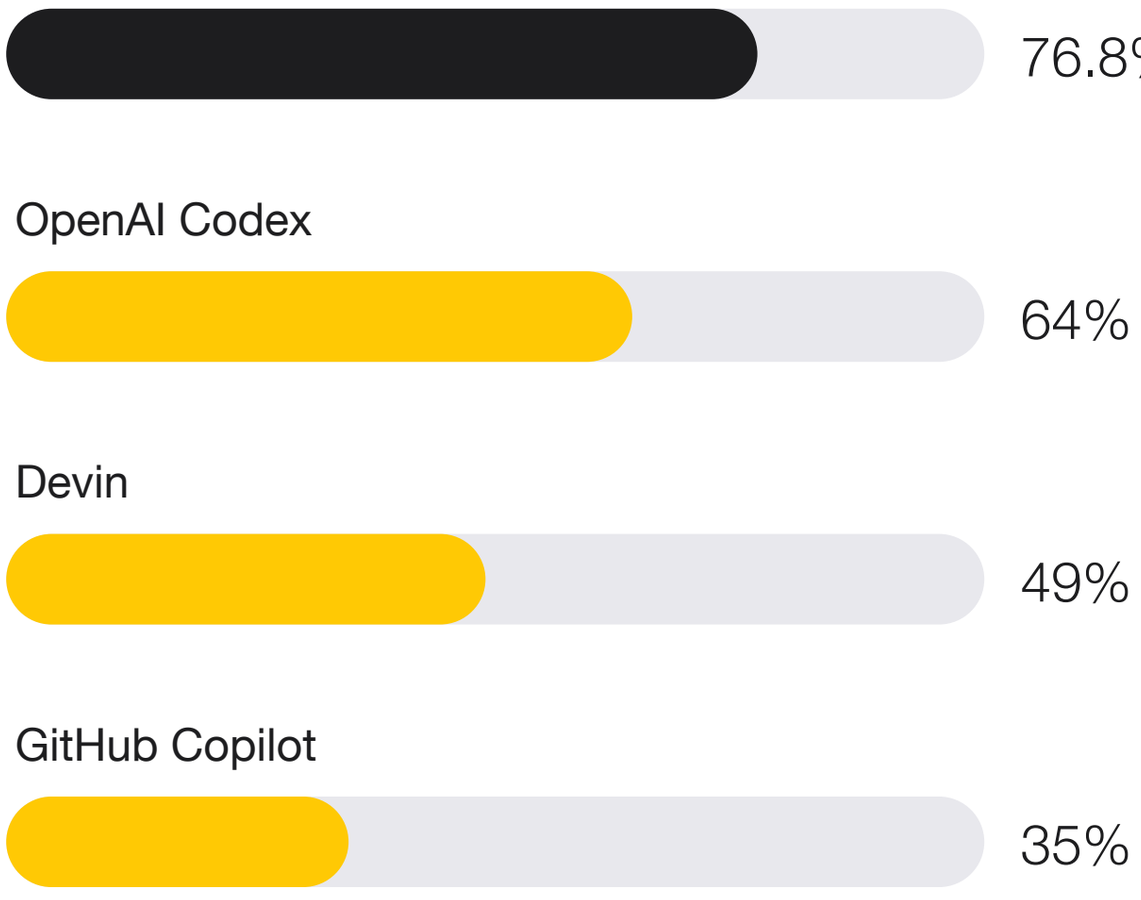
The lesson

“Tests pass” is evidence, not proof. The same is true for your own agent’s PRs.

Agent Pull Requests on GitHub

PR acceptance rate

AIDev, 456K+ agent-authored PRs



9.1%

of agent PRs change code complexity, vs. 23.3% of human PRs

Fast, but routine.

Agents mostly make simpler changes, and they are accepted less often than humans.

Test Coverage of Agent PRs

Our ICSME 2026 study of 4,882 agentic pull requests

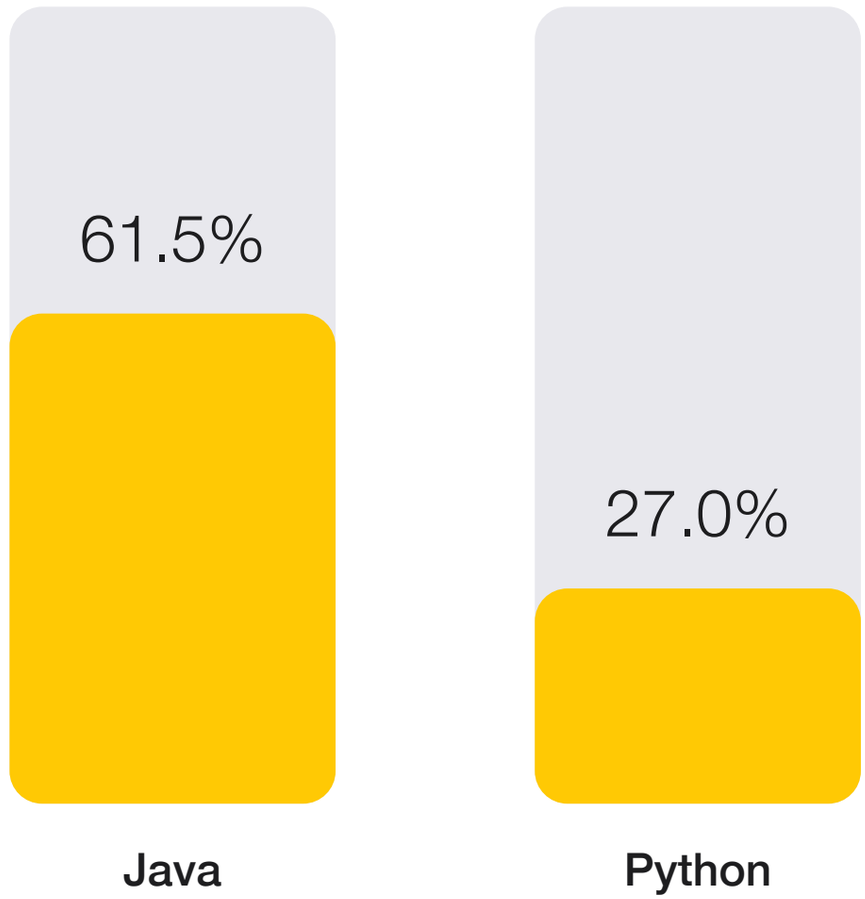
50.4%

of PRs that change code under test include **no test changes**

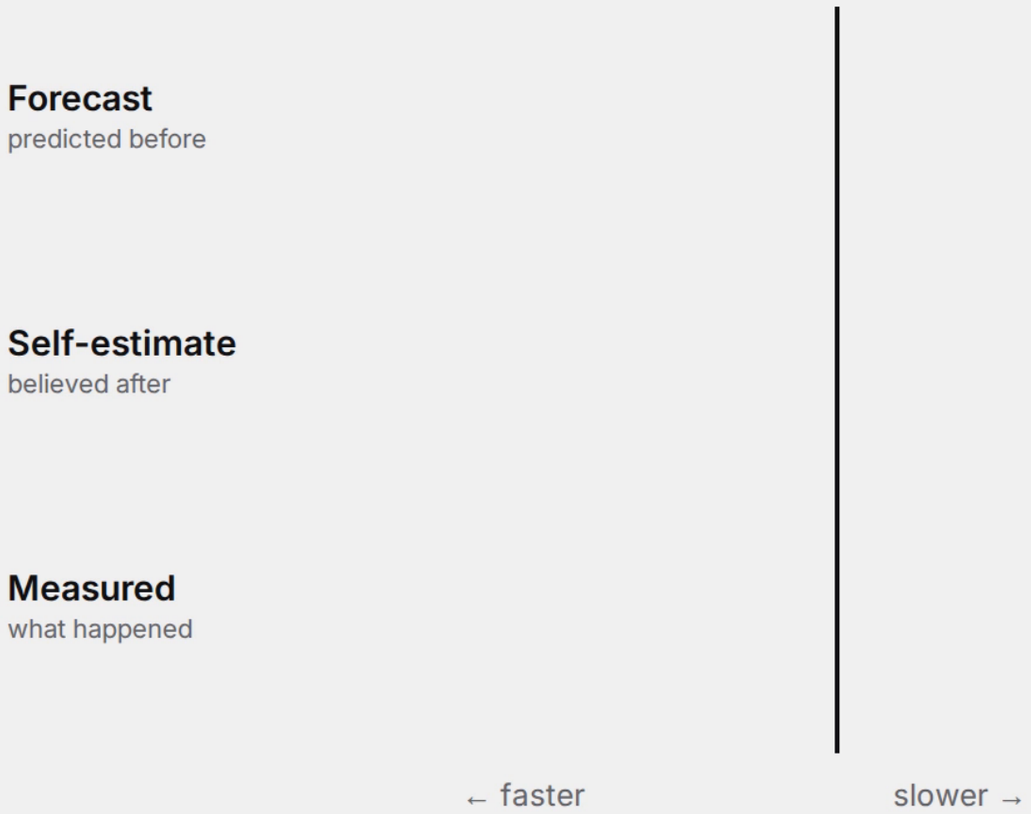
81–86%

of error-handling lines agents add are **never run by tests**

Agent-changed lines run by existing tests



AI and Developer Productivity



+19%

slower on average
while developers believed they were 20% faster

Why? Large, familiar codebases, high quality bars, and time spent prompting, waiting, and fixing output.

Source: Becker et al., METR, 2025

Levels of Confidence in Code

“Done!”	Tests pass	Correct	Understood
the agent says so	automated checks	does what was needed	your team can own it
no evidence	*weak tests fool it*	*needs review and a spec*	*needs you*

STRONGER CLAIM, HARDER TO CHECK

Benchmarks and most agents
stop at “tests pass.”

Your job as the engineer
is “correct” and “understood.”

Deciding What to Delegate

Cost if it slips through low → high

Delegate with tests

fast to catch, costly if missed

- Refactoring with clear rules
- A well-specified, scoped task

Keep or review deeply

slow to surface, costly

- Architecture decisions
- Security-critical code
- Business logic with unstated rules
- Requirements clarification

Delegate freely

fast to catch, cheap

- Boilerplate and repetitive code
- Test scaffolding you review
- Searching a large codebase

Delegate, then spot-check

slow to surface, cheap

- API docs and comments
- Code explanations

How long until a wrong answer surfaces? fast → late

Adapted from O. Chaparro (William & Mary)

Technical Limitations of Agents

Hallucinations

Confident, plausible, and wrong. Worst on obscure libraries, internal conventions, and recent API changes.

Context drift

In long sessions the agent loses the original goal and optimizes for the wrong thing.

Intent blindness

It solves what you said, not what you meant. Vague specs compound.

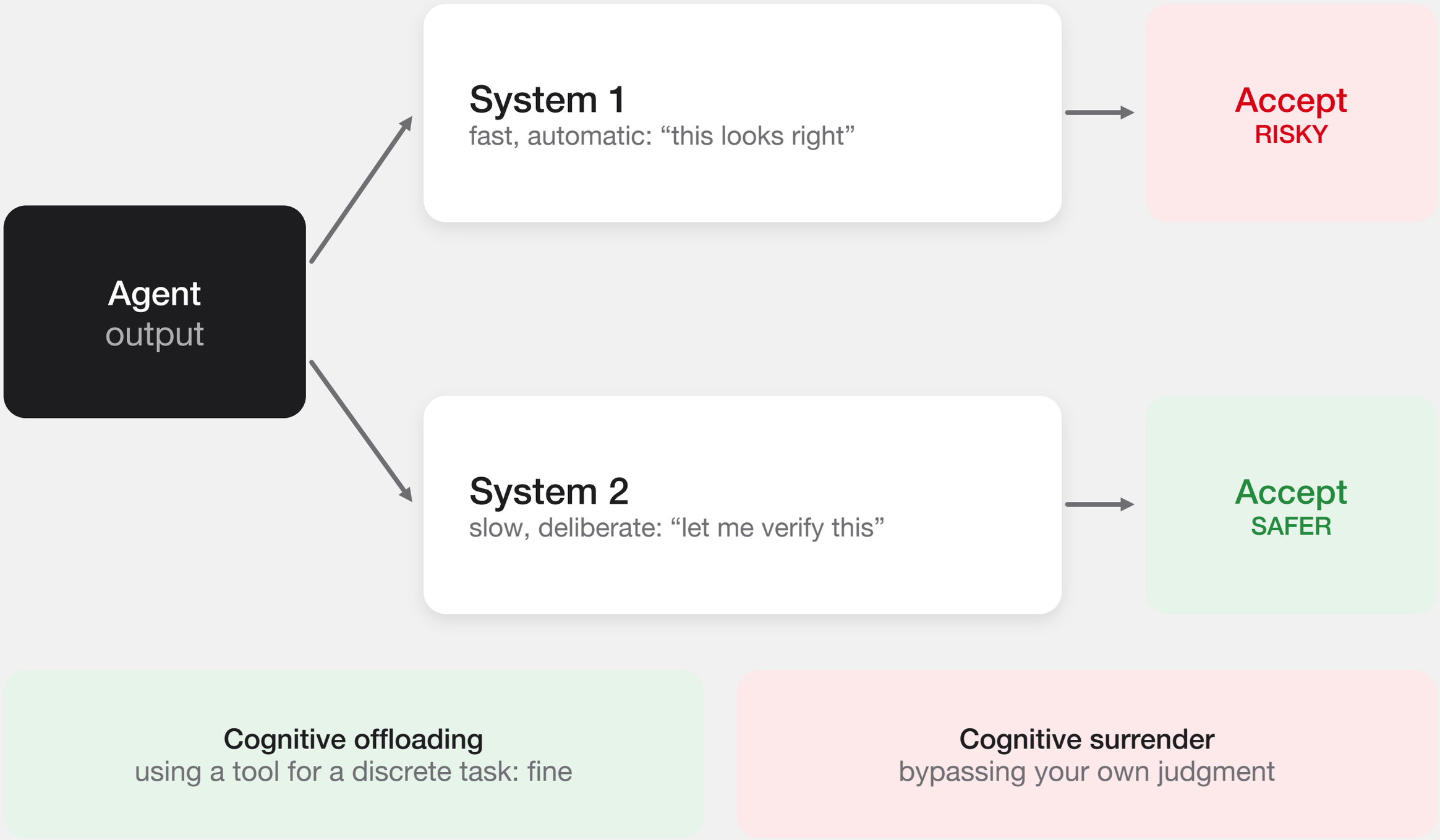
Context limits

It cannot hold a large codebase at once, so it must choose what to read, and may choose wrong.

Common cause: thin training data, and a second model shares the same blind spots.

Adapted from O. Chaparro (William & Mary)

Cognitive Surrender



Cognitive offloading
using a tool for a discrete task: fine

Cognitive surrender
bypassing your own judgment

Adapted from O. Chaparro (William & Mary)

Technical, Cognitive, and Intent Debt

AI may reduce technical debt **while accelerating cognitive and intent debt.**

Intent debt

lives in artifacts

goals, constraints, and decisions never written down

Technical debt

lives in the code

messy code, shortcuts, compromises

amplify / repair

Cognitive debt

lives in people

the team's shared understanding erodes

Adapted from O. Chaparro (William & Mary)

The Core Risk of AI-Assisted Coding

The code may work.

The team may not understand it.

And no one wrote down what it was supposed to do.

This is the core risk of AI-assisted development.

Adapted from O. Chaparro (William & Mary)

Key Takeaways

An agent is a model, a loop, and tools

and it is only as good as its context.

Benchmarks and "feelings" overstate correctness

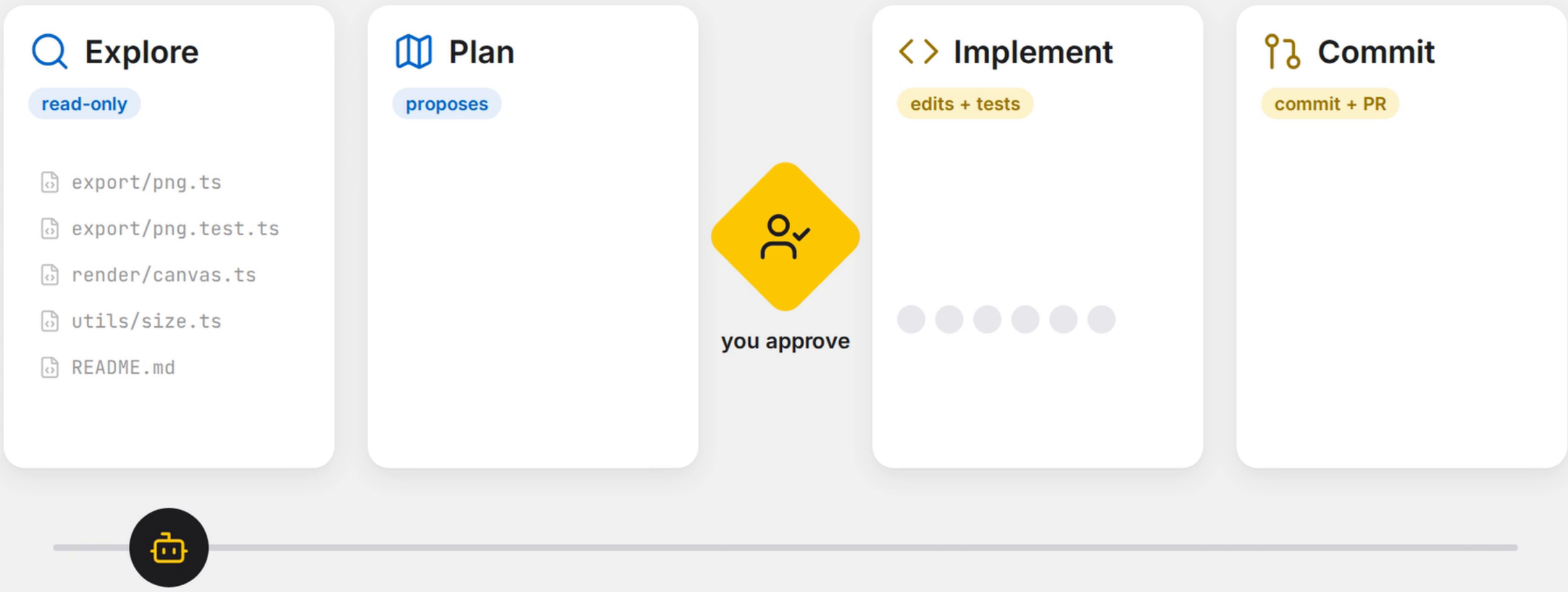
“Tests pass” and “I feel faster” are not proof. Measure and verify.

Delegate by risk

How fast would a mistake surface, and what would it cost?

Next time: workflows, verification, security, and responsibility

Explore, Plan, Implement, Commit



Adapted from O. Chaparro (William & Mary)

Planning Before Implementation

```
# plan.md · gitea/modules/indexer/code

## Open questions
1. Cache results, or recompute each page?
2. Is there a pagination helper already?

## Answers (yours)
1. Recompute for now.
2. Yes: reuse the one in pagination.go.

## Plan
1. Add a regression test for page 2.
2. Trace where the page param is dropped.
3. Fix it with the shared helper.
4. Re-run the indexer test suite.
```

Explore, then plan

Surface the plan for review before any file changes.

Questions go to you

Open questions are requirements. You answer them.

Write it down

A committed plan file survives the session.

Plan mode in Claude Code

Shift + Tab — read-only: explore and propose

Adapted from O. Chaparro (William & Mary)

Vague vs. Scoped Prompts

Vague

"Fix the search bug."

Which bug? Where? How will you know it is fixed?

Scoped

"In **modules/indexer/code**, **results do not paginate past page 1 for repos with 10,000+ files**. **Reproduce with the demo dataset**, **find where the page parameter is dropped and fix it**, and **add a test covering page 2 of a large result set**."

Where | Symptom | How to reproduce | The task | Done when…

A good prompt reads like a good issue: where, what is wrong, how to see it, what to do, and how you will know it is done.

Adapted from O. Chaparro (William & Mary)

Spec-Driven Development

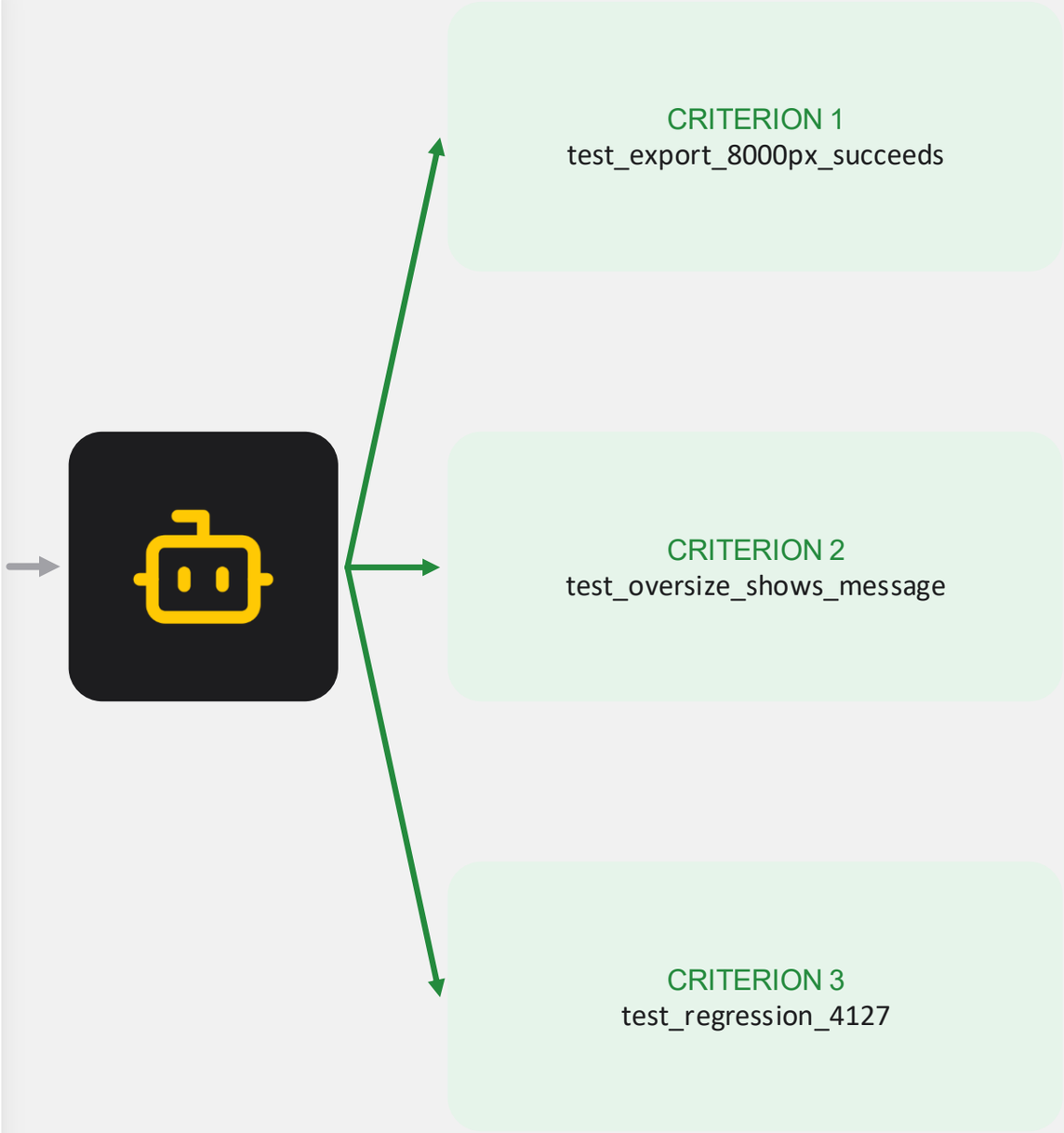
Open #4127 · excalidraw/excalidraw

Export fails for canvases > 4000×4000 px

Steps: open canvas, draw, export to PNG
Expected: export succeeds, or a clear size-limit error

ACCEPTANCE CRITERIA

1. Exports up to 8000×8000 px succeed.
2. Larger canvases show an explicit message.
3. A regression test covers both cases.



Adapted from O. Chaparro (William & Mary)

Project Instruction Files

```
# CLAUDE.md: project notes for AI agents

- Run `go test ./...` before
  opening a PR.

- Follow existing error-handling
  patterns in modules/; no new
  error types without discussion.

- If execution diverges from the
  approved plan, stop and
  re-enter plan mode.

- Never modify .github/workflows/
  without flagging it in the PR.
- Auth or secrets: flag and wait
  for human review.
```

How to verify

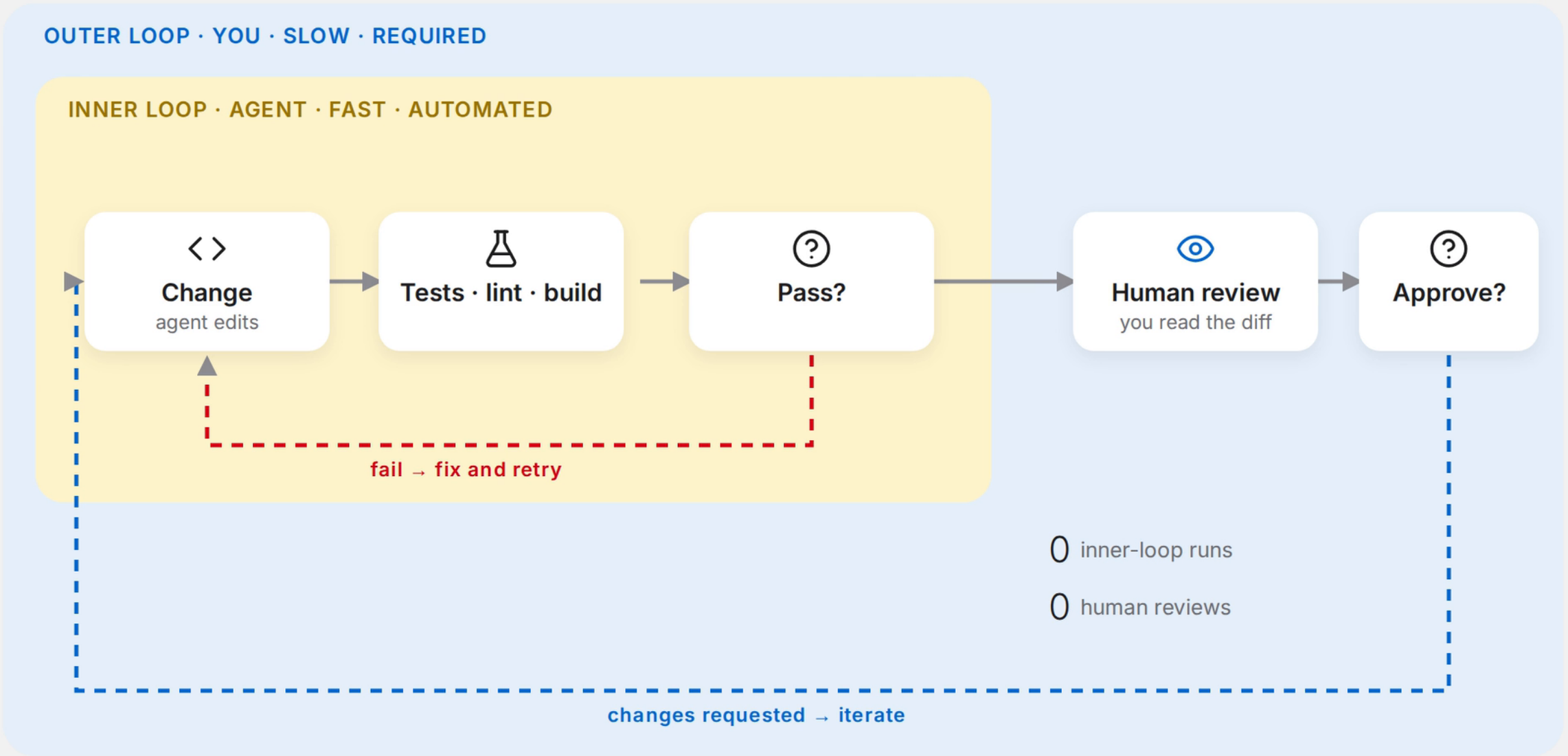
Conventions

Process rules

Guardrails

Adapted from O. Chaparro (William & Mary)

Automated and Human Verification



Task complete ≠ correct ≠ understood. Humans must review security-critical code, auth, and anything hard to reverse.

Adapted from O. Chaparro (William & Mary)

Common Failure Patterns in Agent PRs

Weakened test

changes the check, not the code

```
- expect(total).toBe(10.05)
+ expect(total).toBeDefined()
```

Swallowed error

the failure hides; the bug stays

```
+ try { exportPng(scene) }
+ catch (e) { /* ignore */ }
```

Special-casing

passes this test and nothing else

```
+ if (w === 4001) return ok()
```

Scope creep

unrelated files reformatted

```
142 files changed
+3,905  −3,611
```

Invented dependency

a package that doesn’t exist yet

```
+ import { tile } from
+   "canvas-tile-utils"
```

“Done!” without proof

claims tests pass; never ran them

```
✓ All tests pass!
(no test command in log)
```

Reviewing an Agent's Pull Request

fix(export): guard oversized canvases #4127

src/export/png.ts	+18 −3	3
src/export/png.test.ts	+4 −6	1
src/utils/format.ts	+140 −139	4
package.json	+1	5

```
png.test.ts
- expect(() => exportPng(big))
-   .toThrow(SizeLimitError)
+ expect(exportPng(big))
+   .resolves.toBeTruthy()
  // why did the expectation flip?
```

1. Did the tests change? Read them first.
2. Is every changed line run by a test?
3. Error paths: what happens when it fails?
4. Unrelated files? Ask to split or revert.
5. New dependencies: real, needed, trusted?
6. Can you explain it in your own words?

Using Diff Coverage in Review

src/export/png.ts · lines added in this PR

```
1  export async function exportPng(scene) {
2    const { w, h } = bounds(scene);
3    if (w * h > MAX_PIXELS) {
4      return showSizeError(w, h);
5    }
6    try {
7      return await render(scene, w, h);
8    } catch (e) {
9      log.warn("export failed", e);
10     return fallbackRender(scene);
11   }
12 }
```

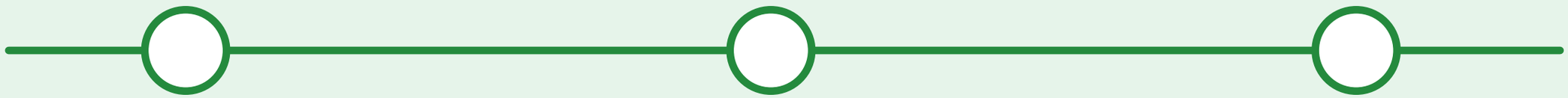
0%

diff coverage

Using Git Commits as Checkpoints

Meaningful checkpoints

bad step? git revert 7d3e81



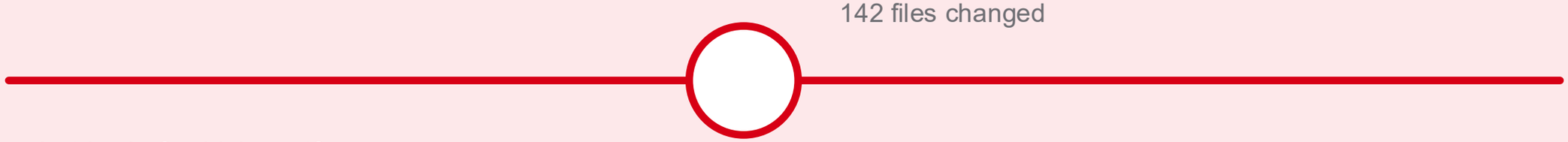
a1c9f2
add failing test for page 2

7d3e81
reuse shared pagination helper

b52a04
guard oversized export, add test

One massive commit

3f0b7c fix stuff
142 files changed



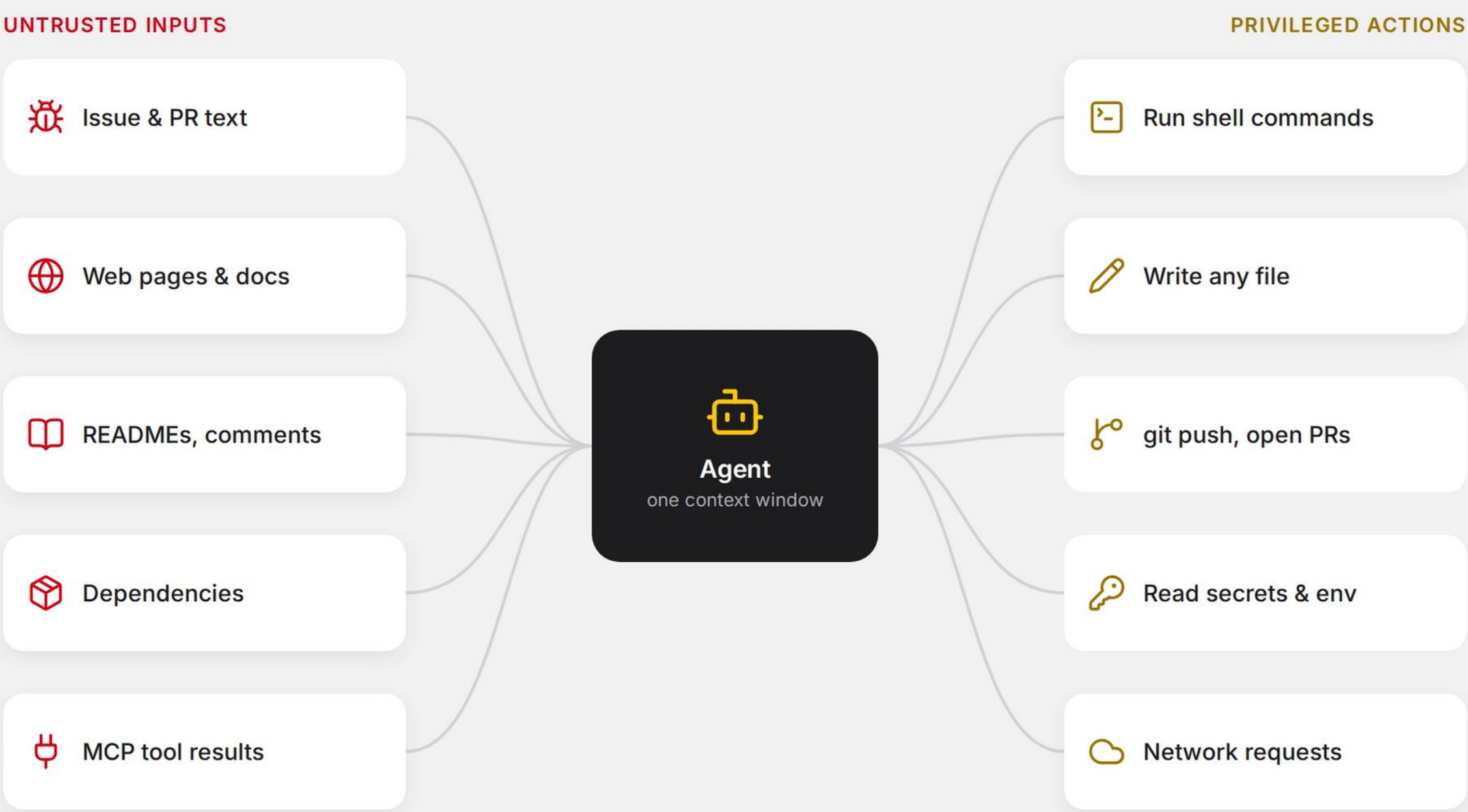
what broke? which part?

CHECKPOINT BEFORE ANYTHING HARD TO UNDO

- schema changes
- auth
- dependencies

Adapted from O. Chaparro (William & Mary)

The Agent's Attack Surface



Case Study: Prompt Injection

1 Attacker plants an issue

2 You ask the agent

3 Agent follows the issue

4 Private data leaks

your-org/public-repo public

Open issues

#51 Crash when saving settings

#54 Docs typo in README

Agent + GitHub MCP

your-org/private-repo

salary.csv

relocation-plan.md

PR #58 “Update README” public

Least privilege

a token scoped to one repo per session

Confirm before writes

no auto-approve for PRs or pushes

Treat inputs as data

be wary when agents read untrusted text

Hallucinated Packages (Slopsquatting)

1 The model invents a package



2 An attacker registers the name

3 Installs pile up

0

downloads

19.7%

of packages recommended by 16 LLMs did not exist

205K+

unique invented package names

16

code-generating LLMs tested

Defense: check that every new dependency exists, is maintained, and is the one you meant. Pin versions.

Least Privilege and Sandboxing



more risk,
more review

Push, deploy, secrets

almost never unattended

Any command, network

installs, web fetches

Run allow-listed commands

tests, lint, build

Edit files in the repo

implement

Read files

explore, plan

.claude/settings.json

```
{
  "permissions": {
    "allow": [
      "Bash(npm run test:*)",
      "Bash(git diff:*)"
    ],
    "deny": [
      "Read(./.env)",
      "Read(./secrets/**)",
      "Bash(git push:*)"
    ]
  }
}
```

Sandbox it: run autonomous sessions in a container or VM with no production credentials.

Harness Extension Points

CLAUDE.md

loaded at start, every session

Hooks

your scripts run on events, every time

Subagents

side tasks in a separate context

session start · your prompt · tool call · tool result · stop

Skills

loaded on demand when relevant

MCP servers

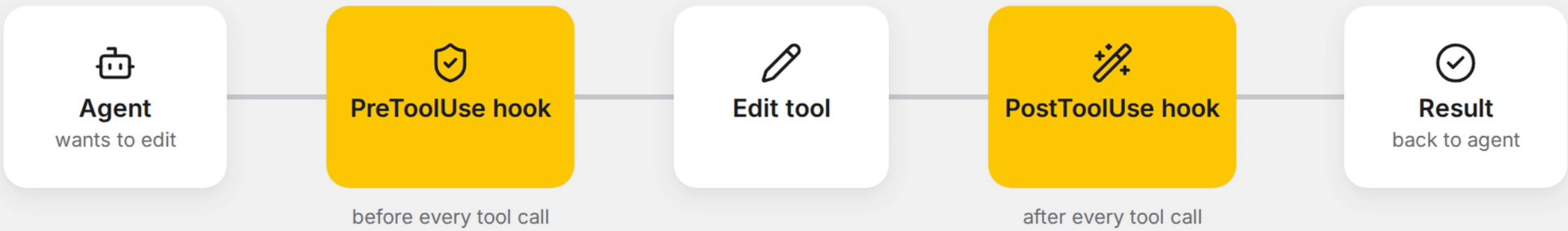
extra tools: GitHub, DBs, browser

Instructions (CLAUDE.md, skills) are advice the model may skip. Hooks and permissions are enforced by the harness.

Hooks: Rules That Always Run

In CLAUDE.md: “Run the formatter after every edit.”
followed… usually

As a hook: npm run format
runs every time




```
"PreToolUse": [{ "matcher": "Edit|Write",
  "hooks": [{ "type": "command",
    "command": "./scripts/protect-ci.sh" }] }]
```

```
"PostToolUse": [{ "matcher": "Edit|Write",
  "hooks": [{ "type": "command",
    "command": "npm run format" }] }]
```

Subagents and Context Isolation



Main session

system + tools

CLAUDE.md

your task

Subagent: test-reviewer

0 files read · tests run

Main context used

.claude/agents/test-reviewer.md

```
---
name: test-reviewer
description: Reviews diffs
  for weakened or missing
  tests. Use after changes.
tools: Read, Grep, Glob, Bash
---
You are a strict test
reviewer. Flag any change
to existing assertions…
```

Also useful: an explorer that maps an unfamiliar module, or a second opinion on a plan.

Skills and MCP Servers

Skills

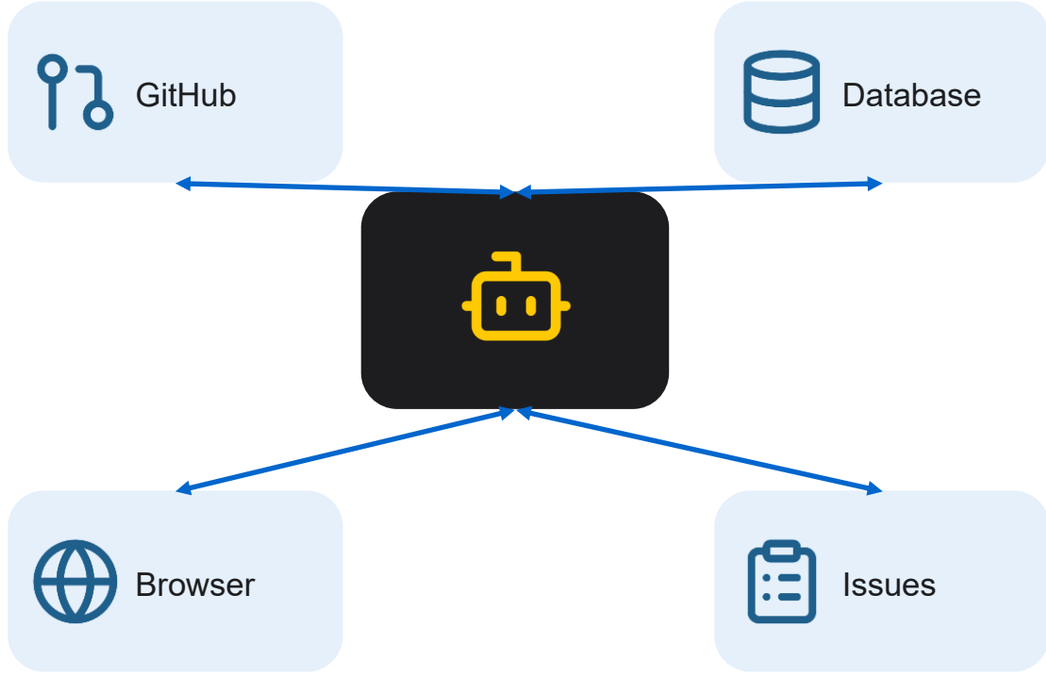
A packaged procedure the agent loads only when the task calls for it.

```
.claude/skills/
  release-checklist/
    SKILL.md
    bump_version.sh
```

- “How we cut a release”
- “How to add a DB migration”

MCP servers

A standard plug for giving the agent new tools and data sources.



Each server adds tools, and attack surface.

Finding Your Project's AI Policy

Ban
Gentoo · NetBSD · QEMU

Disclose and own it
e.g., Ghostty

Like any contribution
no special rules

more restrictive ⟷ more permissive

BEFORE SPRINT 1, FIND YOUR PROJECT'S STANCE

CONTRIBUTING.md

look for an AI or LLM section

AI_POLICY.md

or a policy page in the docs

PR template

is there a disclosure checkbox?

Ask

maintainers, in the project's chat or forum

Accountability for AI-Assisted Work

Agent drafts → You review and test → Your name on it → Maintainers trust you

A DISCLOSURE YOU CAN ADAPT FOR YOUR PR DESCRIPTION

```
## AI assistance
Claude Code drafted the fix and the regression test from the plan in plan.md.
I reviewed every line, ran the full test suite locally, and manually verified
export at 8000×8000 and 9000×9000. The size-limit message wording is mine.
```

✓ Do: understand it, test it, disclose it, keep PRs small

✗ Don’t: submit code you can’t explain, or flood a project with PRs

Practices That Reduce the Three Debts

Intent debt

lives in artifacts

- Issues with acceptance criteria
- Committed plan files
- CLAUDE.md with the “why”
- Disclosure in PRs

Cognitive debt

lives in people

- Explore before you implement
- Explain the diff in your own words
- Write the first test yourself
- Rotate who reviews agent PRs

Technical debt

lives in the code

- Small, scoped diffs
- Checkpoint commits
- Diff coverage on every PR
- Hooks for format and lint

Key Takeaways

Explore → Plan → Implement → Commit

No files change until you approve a plan. Scope prompts like good issues.

Verify, then review

Read the tests first. Check coverage, error paths, scope, and dependencies.

Least privilege

Untrusted input plus privileged action equals risk. Deny the dangerous rungs.

You own what you submit

Follow your project’s policy, disclose, and be able to explain every line.