

CEN 5016:
Software
Engineering

Spring 2024



University of
Central Florida

Dr. Kevin Moran

Week 4 - Class 11:
Static & Dynamic
Analysis





- *Assignment 2 Due Today*
- *Assignment 3 & SDE Project Part 1*
 - Will be posted today
 - Both will be due Thursday February 8th
 - Get started early!!

Intro to Software 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
 - ...

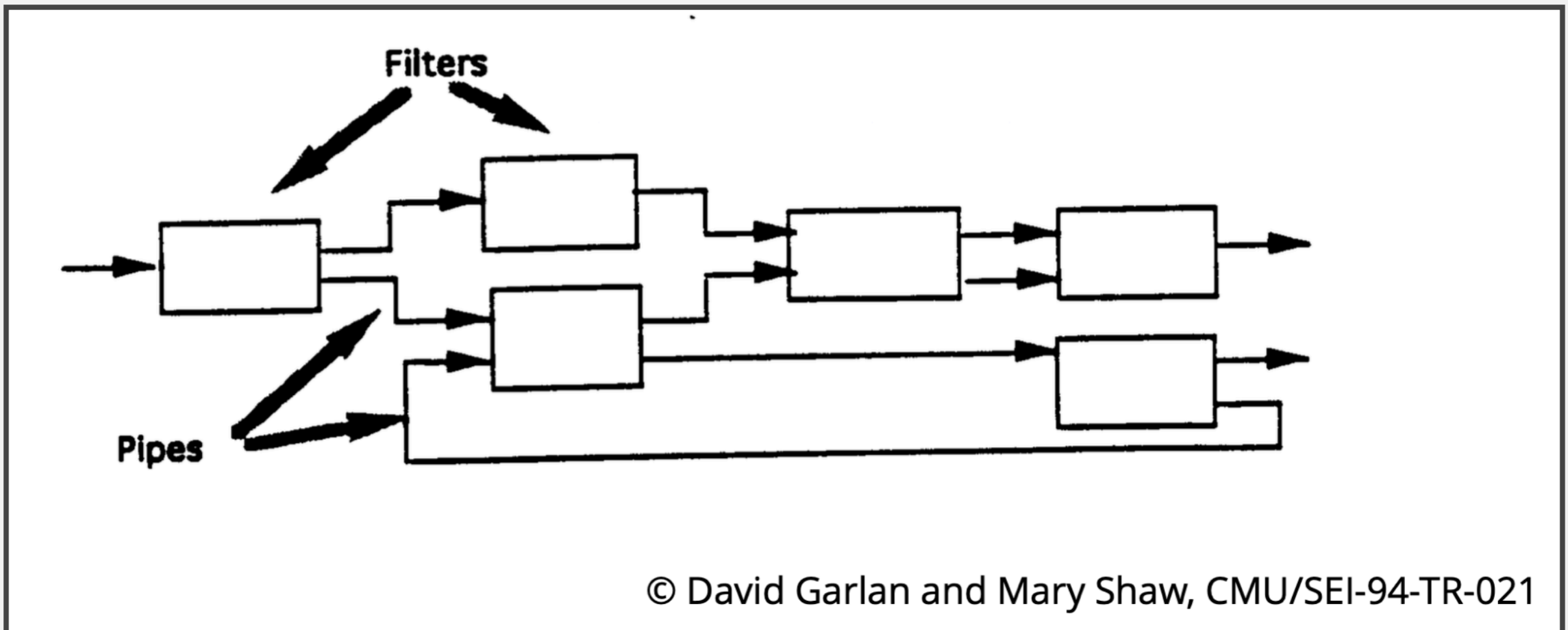


- Static View
 - Modules (subsystems, structures) and their relations (dependencies, ...)
- Dynamic View
 - Components (processes, runnable entities) and connectors (messages, data flow, ...)
- Physical View (Deployment)
 - Hardware structures and their connections

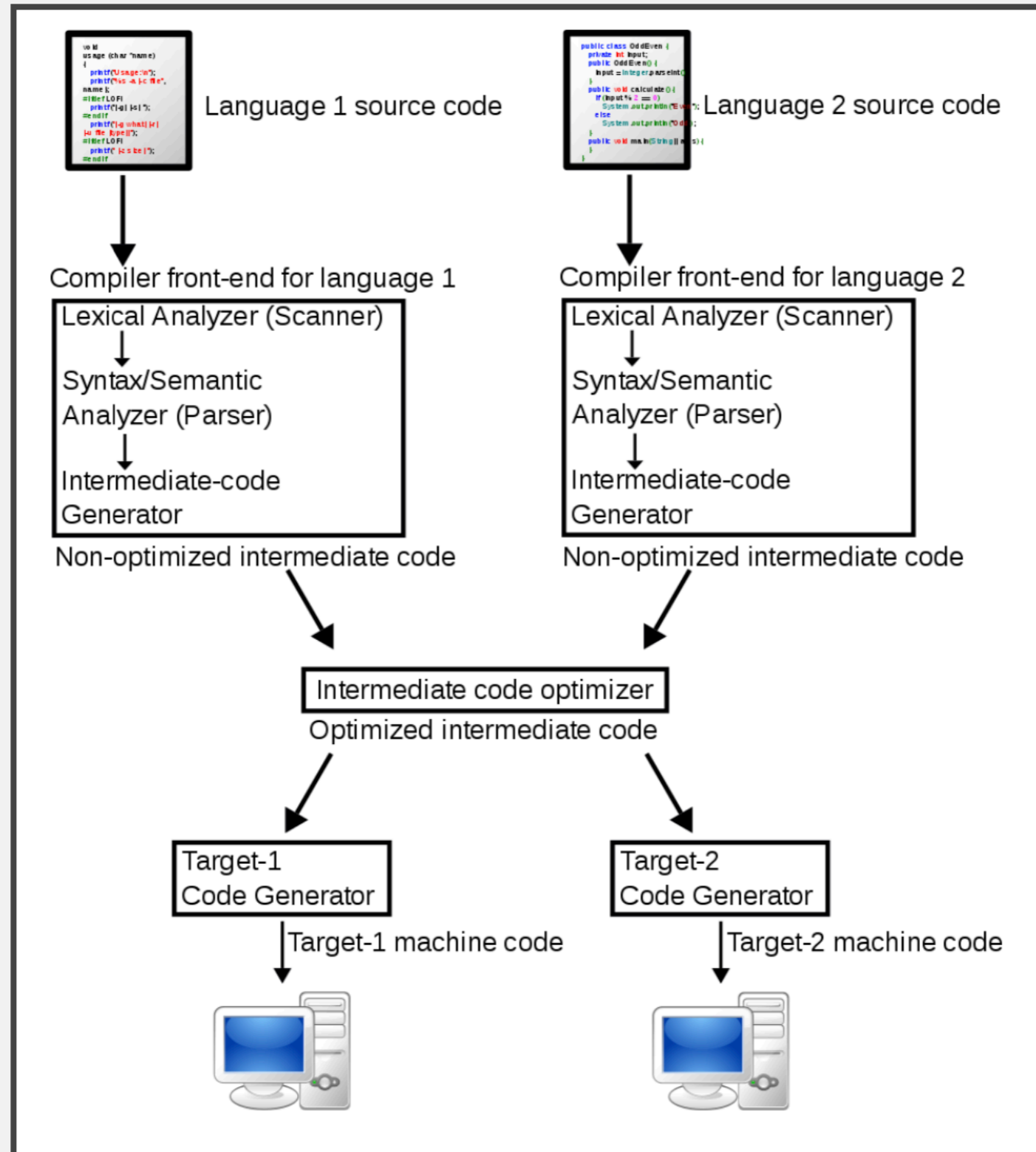
Common Software Architectures



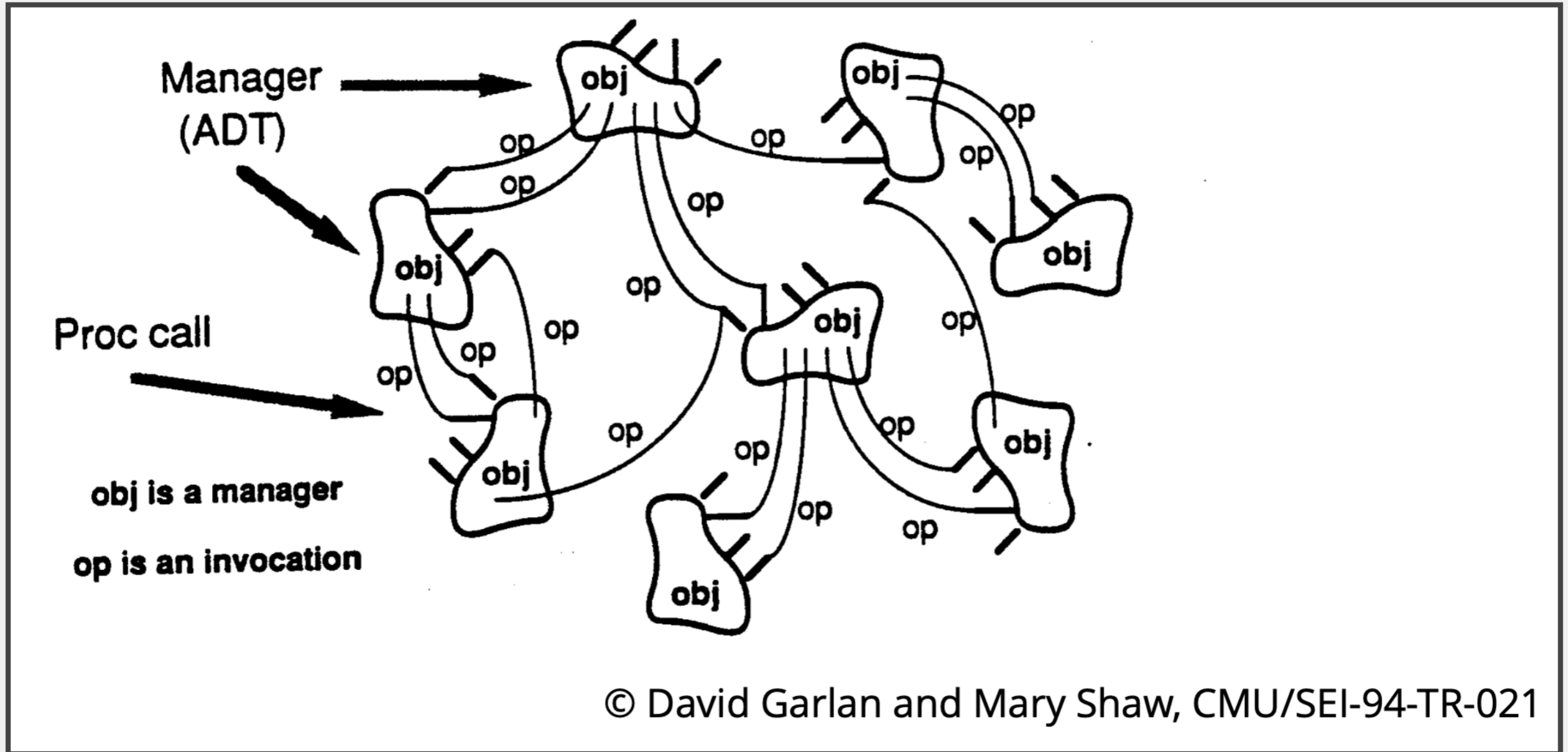
1. Pipes & Filters



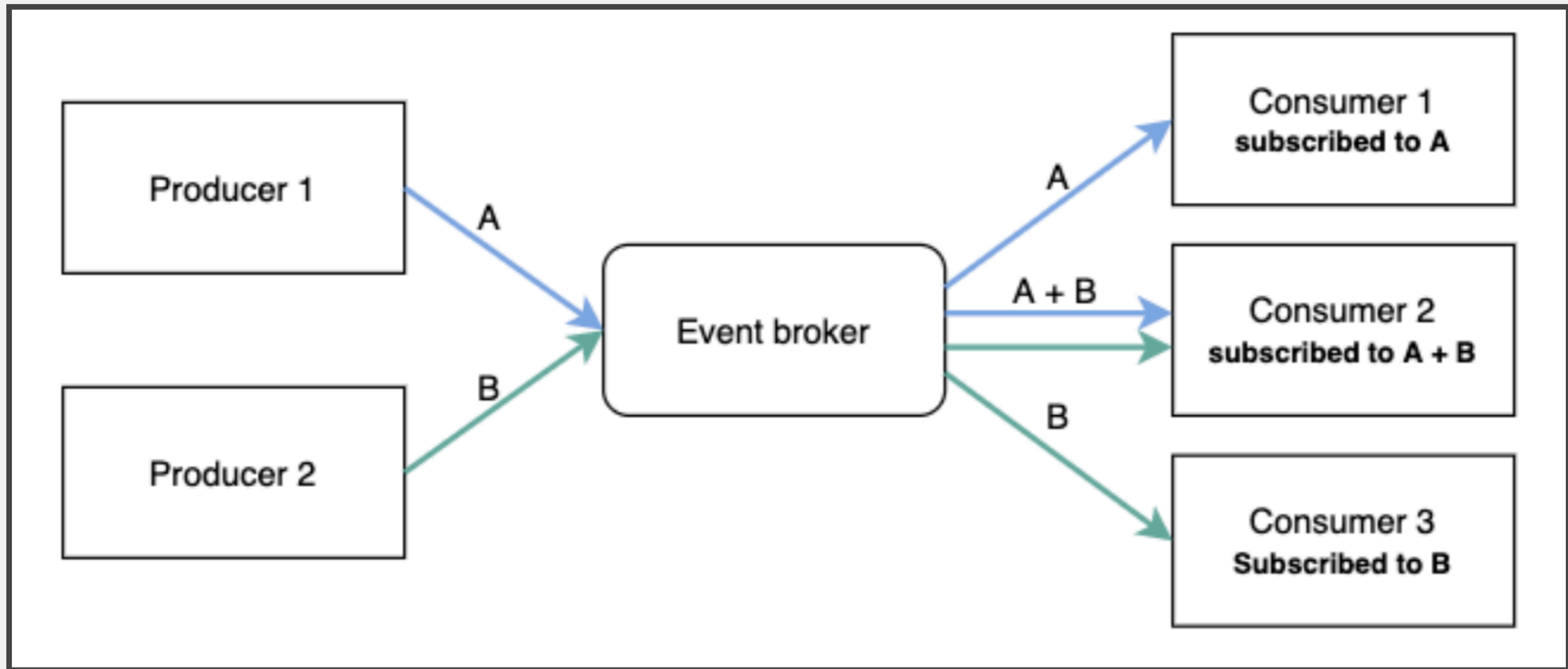
Pipes & Filters Example: Compilers



2. Object Oriented Organization



3. Event-Driven Architecture



Example: HTML DOM + Javascript



NodeBB

Welcome to the demo instance of NodeBB!

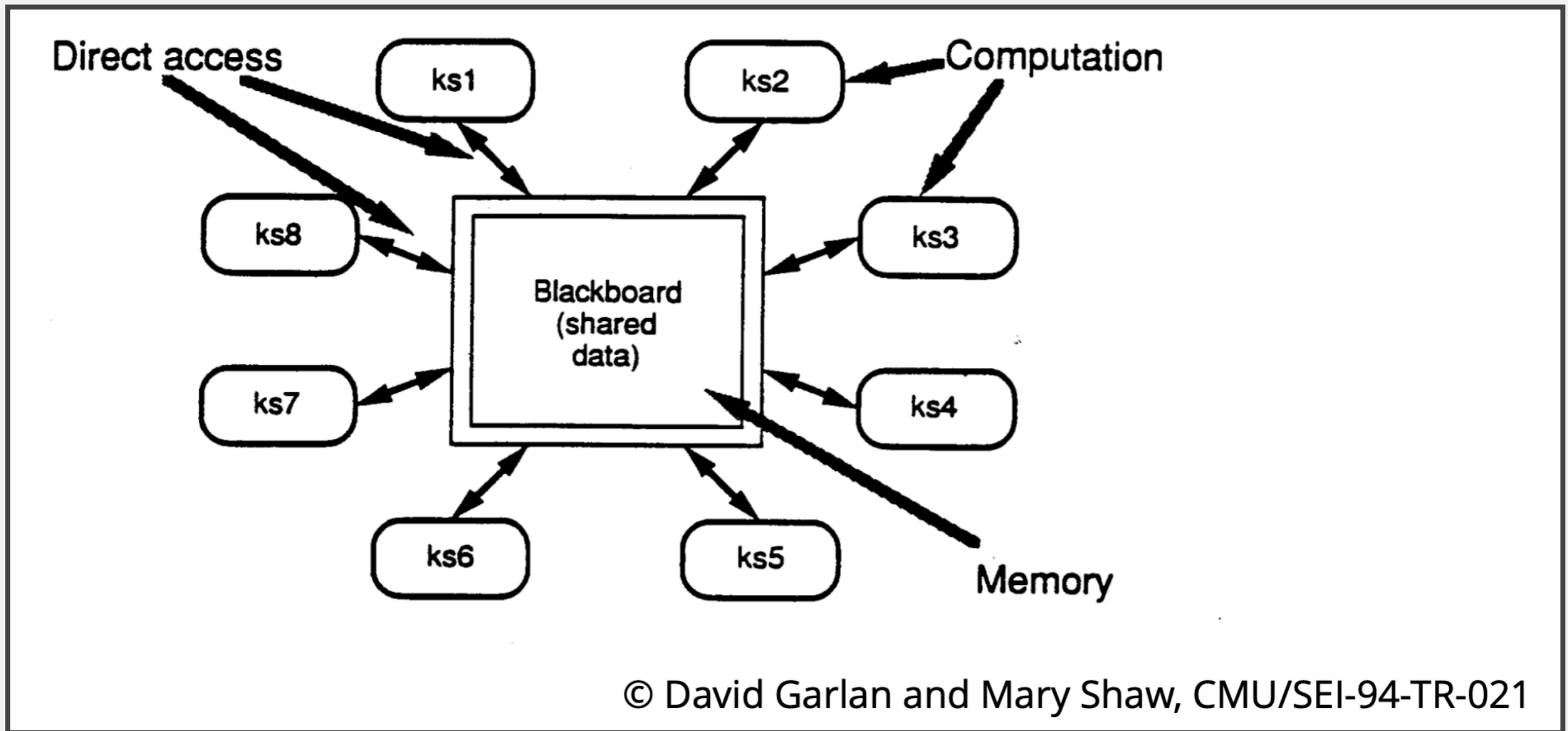
Announcements 1 posts 1 posters 15 views

Sort by 

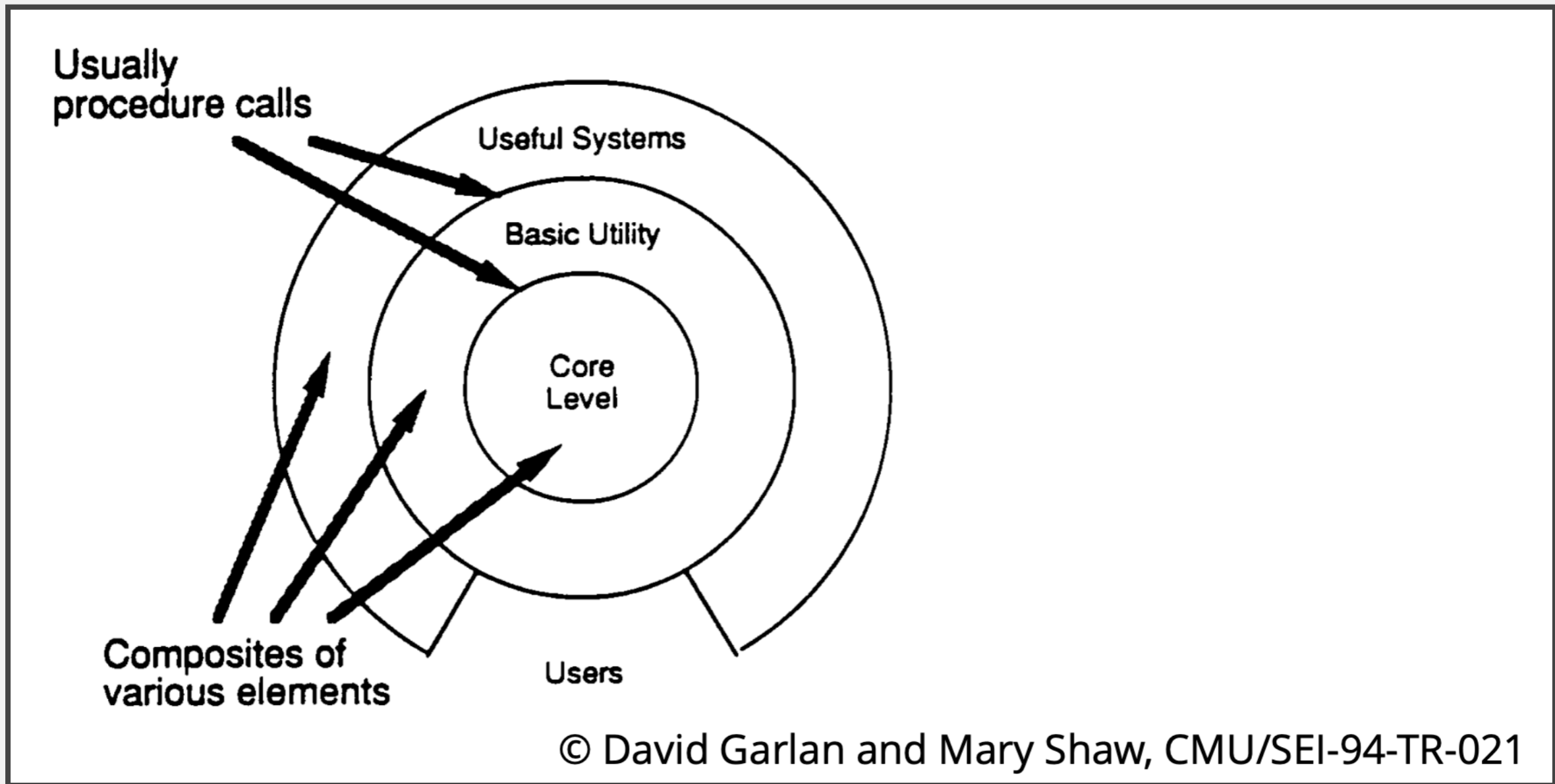
- Oldest to Newest ✓
- Newest to Oldest
- Most Votes

12, 2017, 3:54 PM 

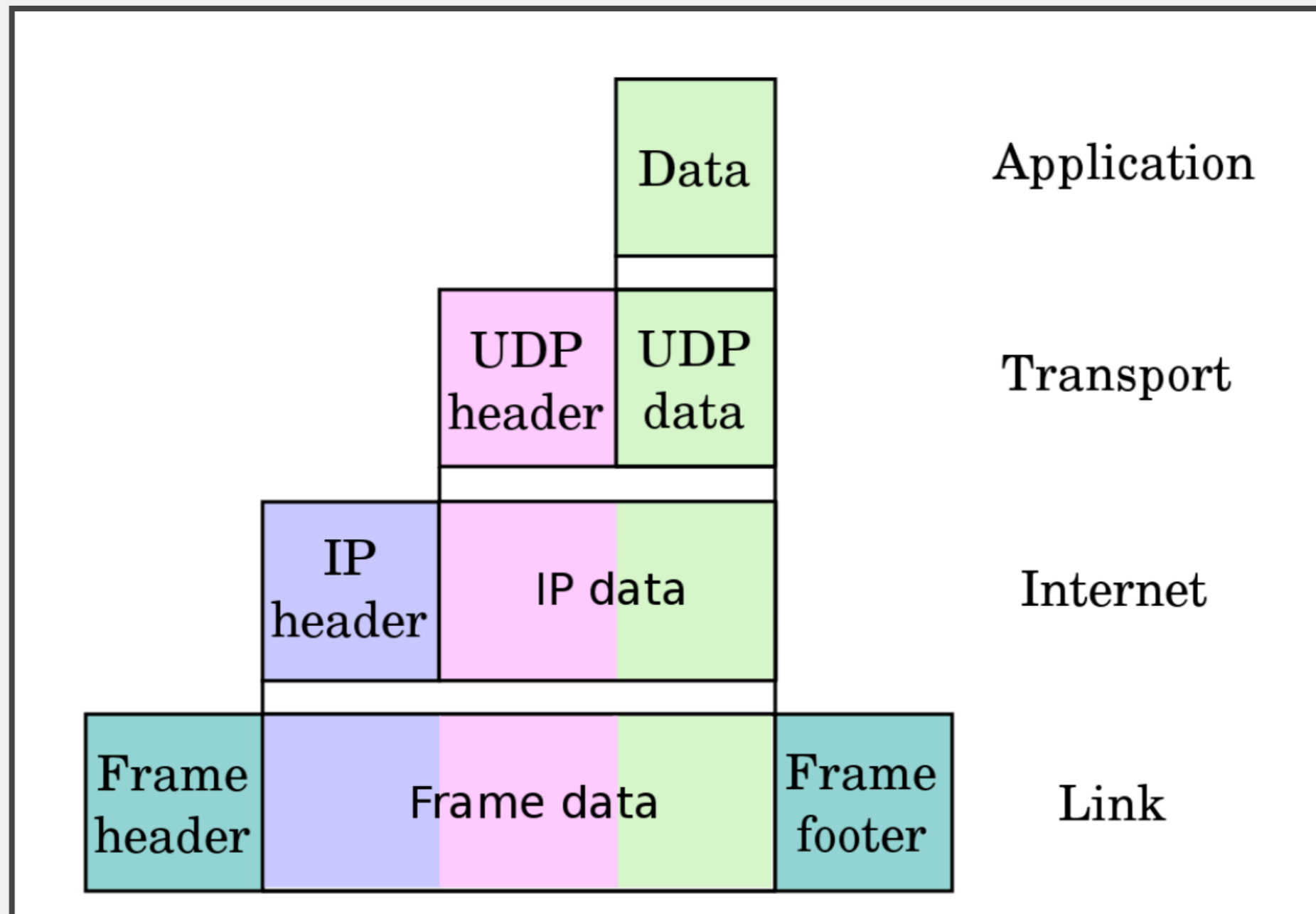
4. Blackboard Architecture



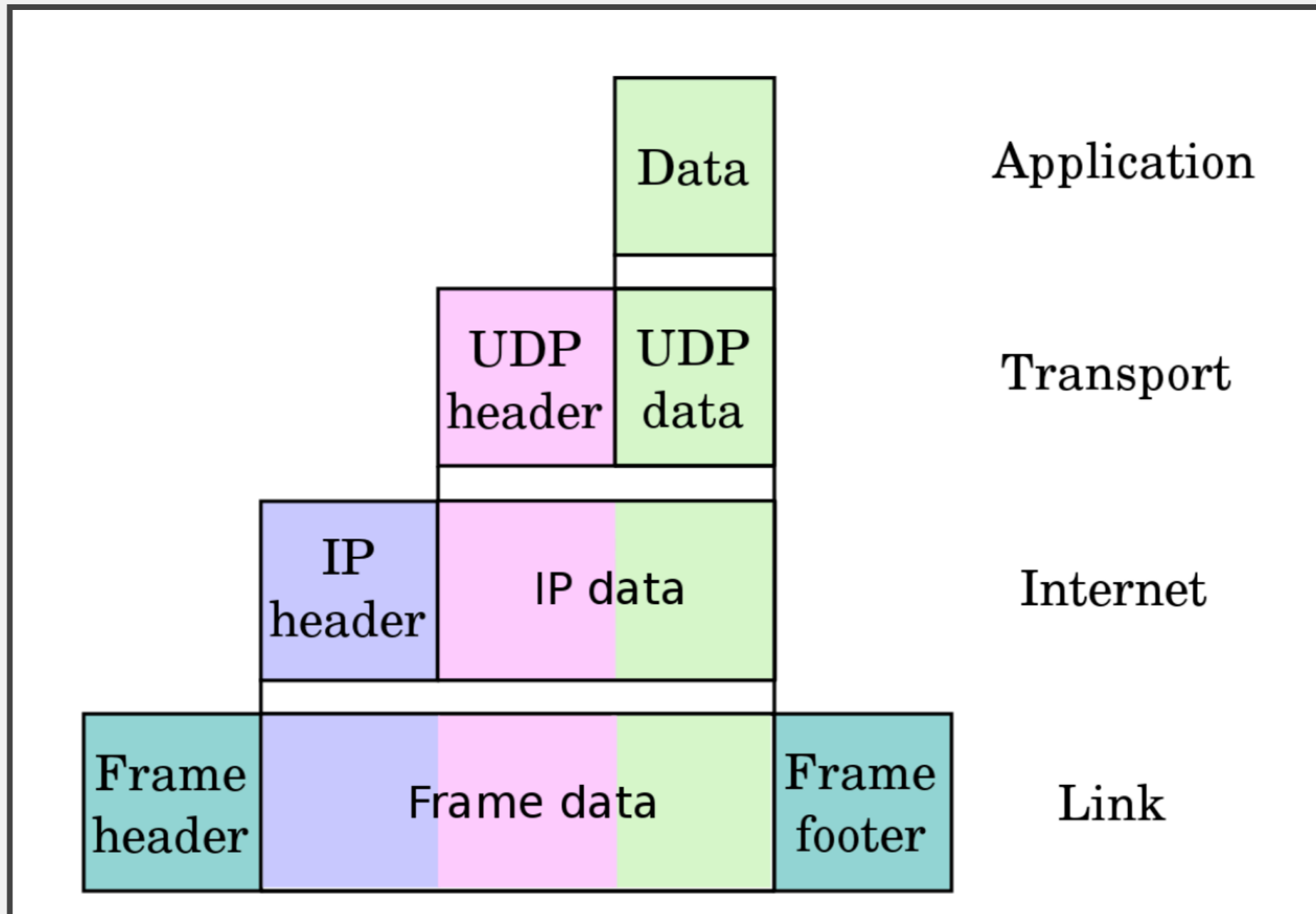
5. Layered Systems



Example Internet Protocol Suite



Example Internet Protocol Suite



Microservices

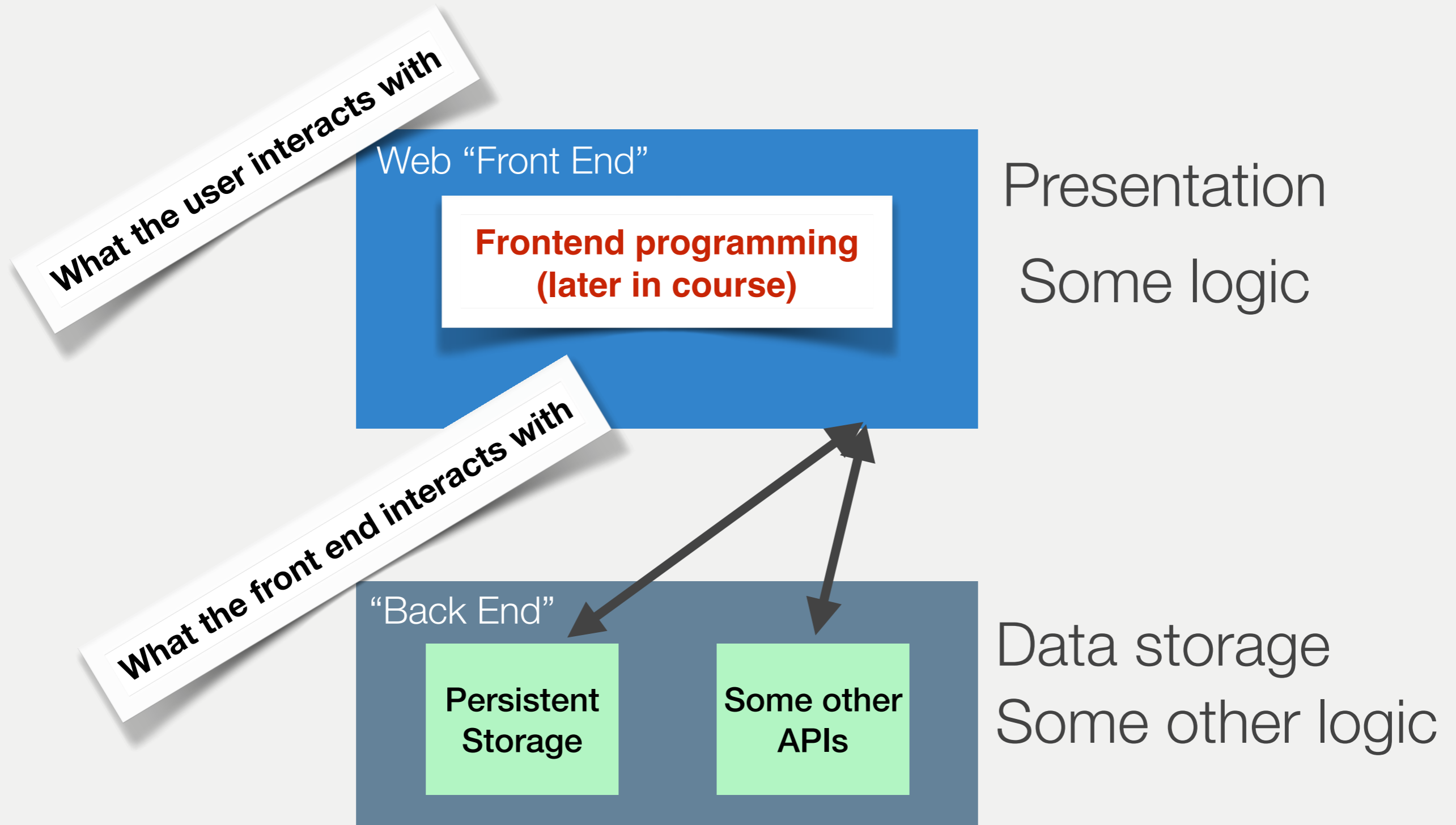




Why We Need Backends

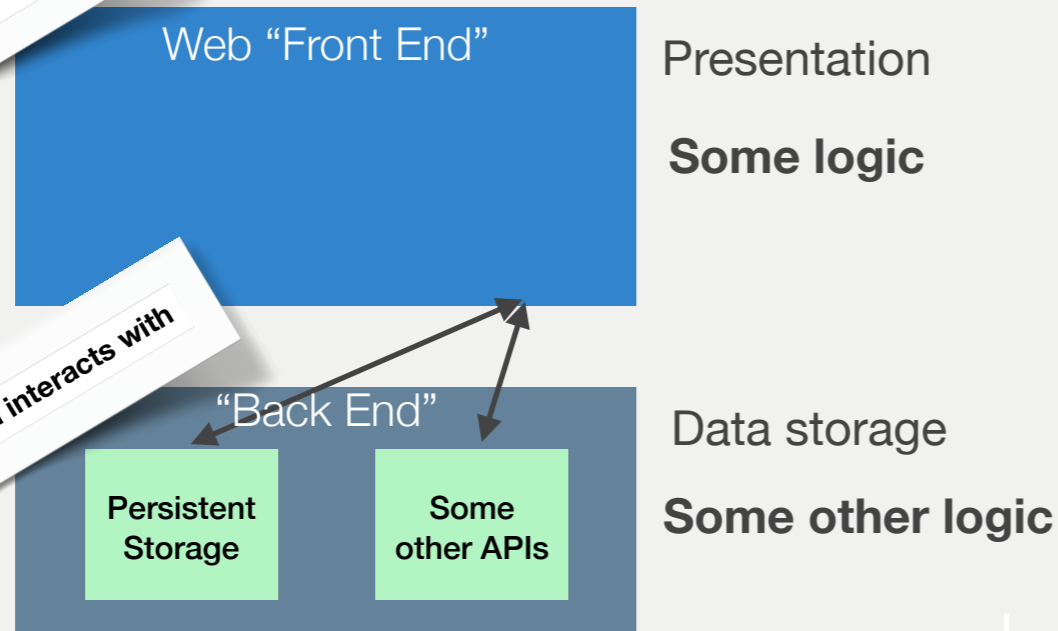
- Security: *SOME* part of our code needs to be “**trusted**”
 - Validation, security, etc. that we don’t want to allow users to bypass
- Performance:
 - Avoid **duplicating** computation (do it once and cache)
 - Do **heavy** computation on more powerful machines
 - Do data-intensive computation “**nearer**” to the data
- Compatibility:
 - Can bring some **dynamic** behavior without requiring much JS support

Dynamic Web Apps





Where Do We Put the Logic?



Frontend Pros

Very responsive (low latency)

Frontend Cons

Security

Performance

Unable to share between front-ends

Backend Pros

Easy to refactor between multiple clients

Logic is hidden from users (good for security, compatibility, etc.)

Backend Cons

Interactions require a round-trip to server

Why Trust Matters

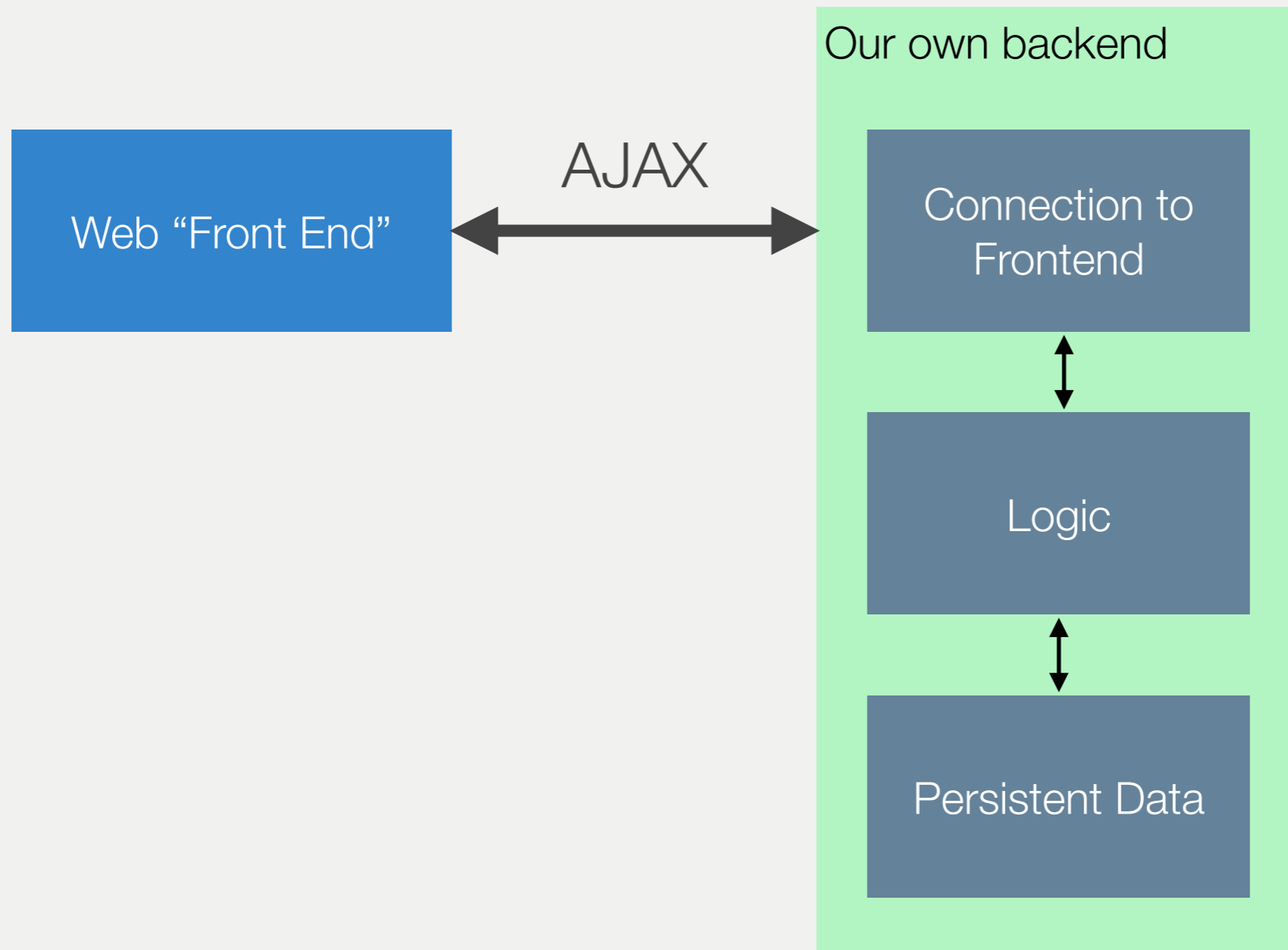


- Example: Banking app
 - Imagine a banking app where the following code runs in the browser:

```
function updateBalance(user, amountToAdd)
{
  user.balance = user.balance + amountToAdd;
}
```

- What's wrong?
- How do you fix that?

What Does our Backend Look Like?





The “Good” Old Days of Backends

HTTP Request

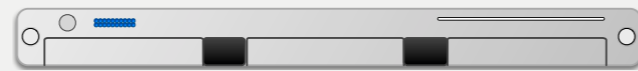
GET /myApplicationEndpoint **HTTP/1.1**

Host: cs.ucf.edu

Accept: text/html

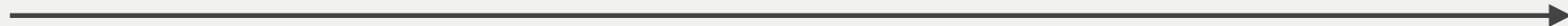
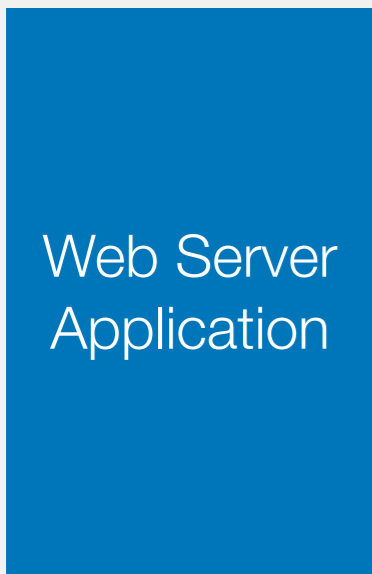


web server



Runs a program

Give me /myApplicationEndpoint



Here's some text to send back



HTTP Response

HTTP/1.1 200 OK

Content-Type: text/html; charset=UTF-8

<html><head>...

What's wrong with this picture?



History of Backend Development

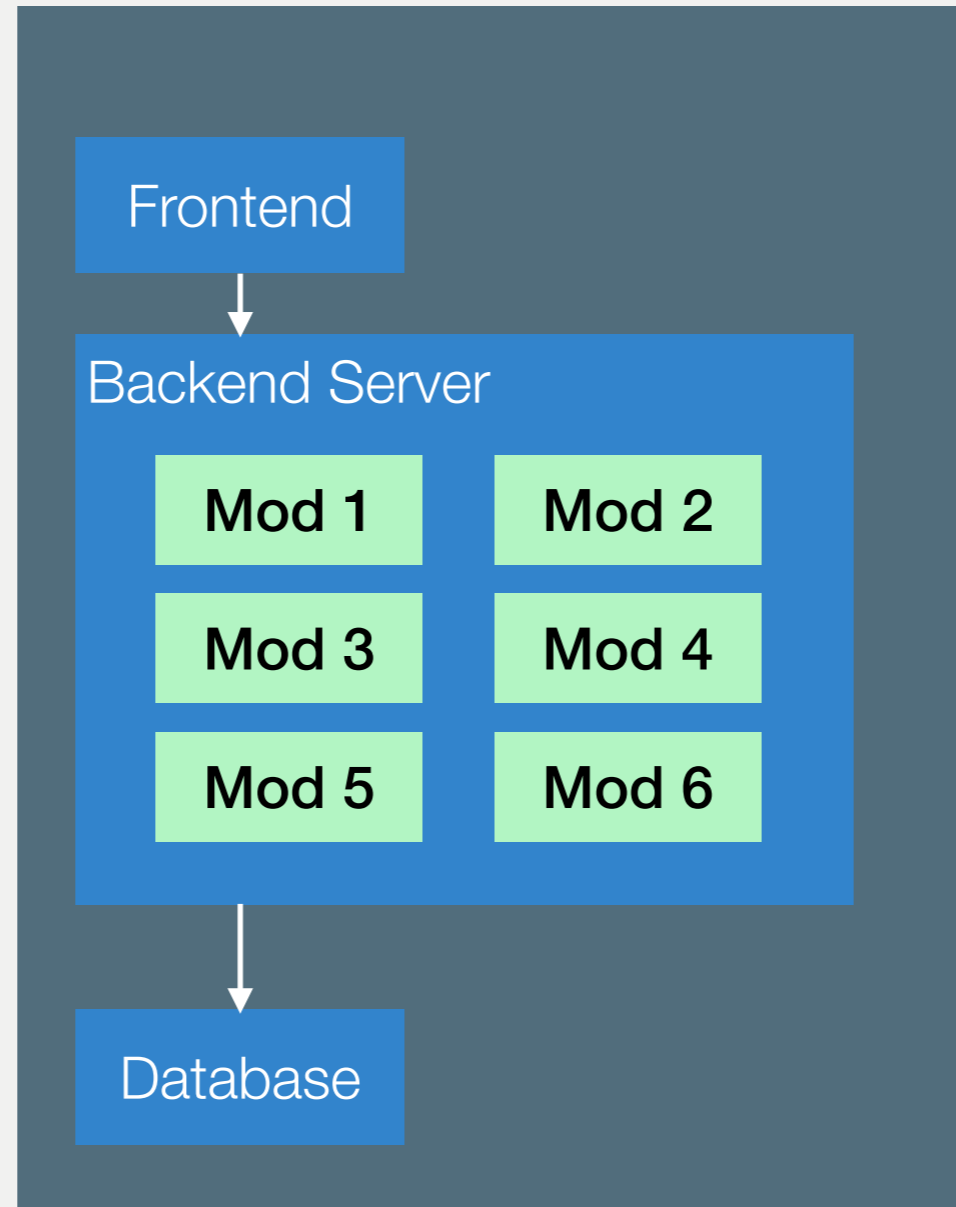
- In the beginning, you wrote whatever you wanted using whatever language you wanted and whatever framework you wanted
- Then... PHP and ASP
 - Languages “designed” for writing backends
 - Encouraged spaghetti code
 - A lot of the web was built on this
- A whole lot of other languages were also springing up in the 90's...
 - Ruby, Python, JSP



Microservices vs. Monoliths

- Advantages of microservices over monoliths include
 - Support for scaling
 - Scale vertically rather than horizontally
 - Support for change
 - Support hot deployment of updates
 - Support for reuse
 - Use same web service in multiple apps
 - Swap out internally developed web service for externally developed web service
 - Support for separate team development
 - Pick boundaries that match team responsibilities
 - Support for failure

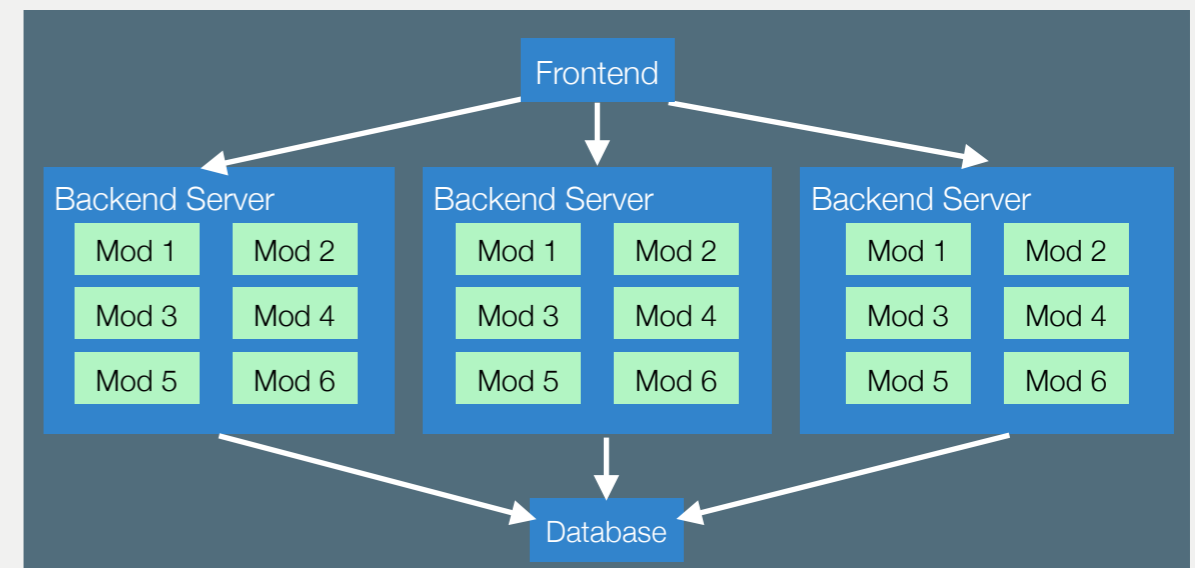
Support for Scaling



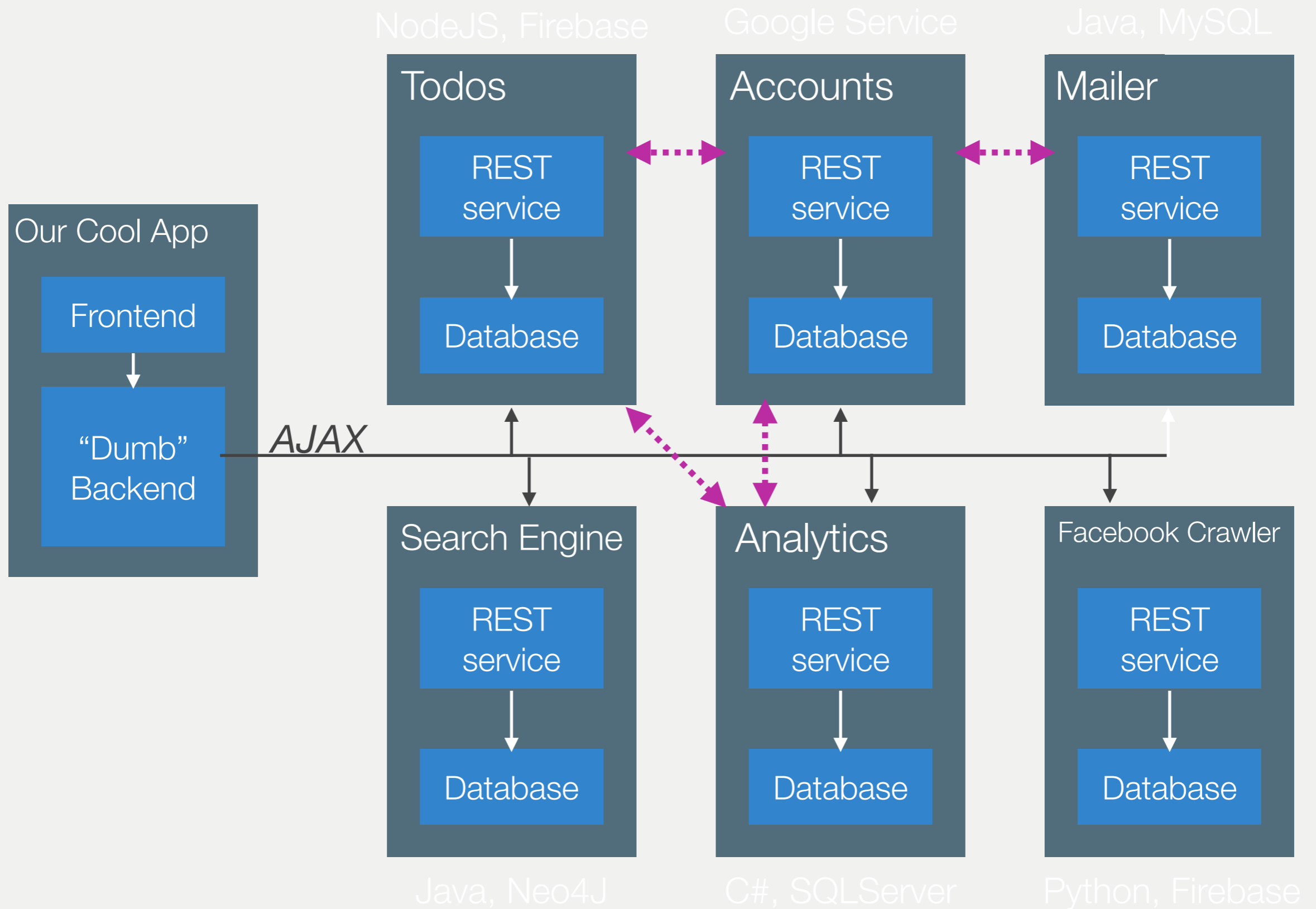


What's wrong with this picture?

- This is called the “monolithic” app
- If we need 100 servers...
- Each server will have to run EACH module
- What if we need more of some modules than others?



Microservices



Goals of Microservices



- Add them independently
 - Upgrade the independently
 - Reuse them independently
 - Develop them independently
-
- ==> Have ZERO coupling between microservices, aside from their shared interface

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

Software QA: Static & Dynamic Analysis





- Gain an understanding of the relative strengths and weaknesses of static and dynamic analysis
- Examine several popular analysis tools and understand their use cases
- Understand how analysis tools are used in large open-source software

Activity: Analyze the Python Program Staticly



```
def n2s(n: int, b: int) -> str:
    if n <= 0:
        return '0'
    r = ""
    while n > 0:
        u = n % b
        if u >= 10:
            u = chr(ord('A') + u - 10)
        n = n // b
        r = str(u) + r
    return r
```

1. What are the set of data types taken by variable `u` at any point in the program?
2. Can the variable `u` be a negative number?
3. Will this function always return a value?
4. Can there ever be a division by zero?
5. Will the returned value ever contain a minus sign '-'?

Answer: Yes, No, Maybe

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



- **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



- **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?”)



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



```
def n2s(n: int, b: int) -> str:
    if n <= 0:
        return '0'
    r = ""
    while n > 0:
        u = n % b
        if u >= 10:
            u = chr(ord('A') + u - 10)
        n = n // b
        r = str(u) + r
    return r
print n2s(12, 10))
```

Answer: Yes, No, Maybe

1. What are the set of data types taken by variable `u` at any point in the program?
2. Did the variable `u` ever contain a negative number?
3. For how many loop executions did the while loop execute?
4. Was there a division by zero?
5. Did the returned value ever contain a minus sign '-'?



- 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



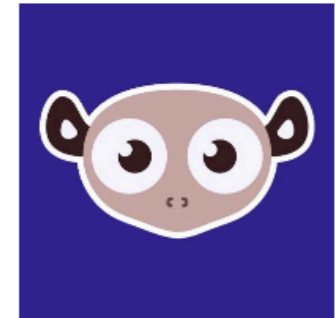
- Requires only source code
- Conservatively reasons about all possible
- Reported warnings may contain false positives
- Can report all warnings of a particular class of problems
- Advanced techniques like verification can prove certain complex properties, but rarely run in CI due to cost

- 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

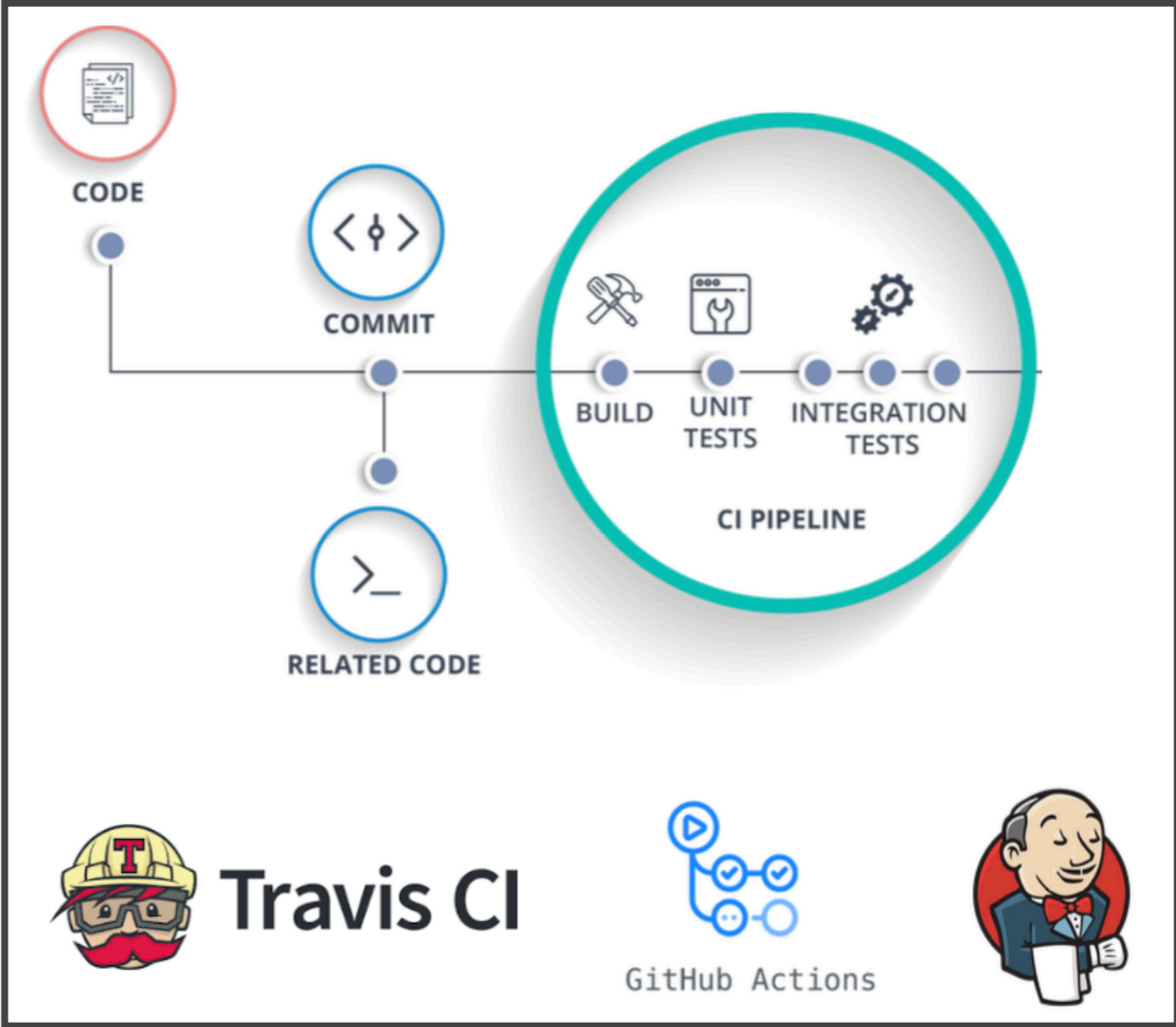
Static Analysis



Tools for Static Analysis



Static Analysis is a Key Part of CI



Static Analysis used to be Purely Academic...



GitHub acquires code analysis tool Semmle

Frederic Lardinois @frederick / 1:30 pm EDT • September 18, 2019

Comment



Marketplace Search results

Types

Apps

Actions

Categories

API management

Chat

Code quality

Code review

Continuous integration

Dependency management

Deployment

IDEs

Learning

Localization

Mobile

Monitoring

Project management

Publishing

Search for apps and actions

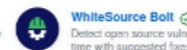
Apps

Build on your workflow with apps that integrate with GitHub.

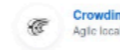
306 results filtered by Apps



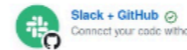
Zube
Agile project management that lets the entire team work with developers on GitHub



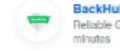
WhiteSource Bolt
Detect open source vulnerabilities in real time with suggested fixes for quick remediation



Crowdin
Agile localization for your projects



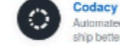
Slack + GitHub
Connect your code without leaving Slack



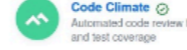
BackHub
Reliable GitHub repository backup, set up in minutes



GitLocalize
Continuous Localization for GitHub projects



Codacy
Automated code reviews to help developers ship better software, faster



Code Climate
Automated code review for technical debt and test coverage



Semaphore
Test and deploy at the push of a button



Flapstastic
Manage flaky unit tests. Click a checkbox to instantly disable any test on all branches. Works with your current test suite



DeepScan
Advanced static analysis for automatically finding runtime errors in JavaScript code



Depfu
Automated dependency updates done right



News

Snyk Secures \$150M, Snags \$1B Valuation



Sydney Sawaya | Associate Editor
January 21, 2020 1:12 PM

Share this article:



Snyk, a developer-focused security startup that identifies vulnerabilities in open source applications, announced a \$150 million Series C funding round today. This brings the company's total investment to \$250 million alongside reports that put the company's valuation at more than \$1 billion.



Static Analysis is Also Integrated into IDEs



```
cppcoreguidelines.cpp x
1 // To enable only C++ Core Guidelines checks
2 // go to Settings/Preferences | Editor | Inspections | C/C++ | Clang-Tidy
3 // and provide: -*,cppcoreguidelines-* in options
4
5 void fill_pointer(int* arr, const int num) {
6     for(int i = 0; i < num; ++i) {
7         arr[i] = 0;
8     }
9     Do not use pointer arithmetic
10
11 void fill_array(int ind) {
12     int arr[3] = {1,2,3};
13     arr[ind] = 0;
14 }
15
16 void cast_away_const(const int& magic_num)
17 {
18     const_cast<int&>(magic_num) = 42;
19 }
20
```

```
97 new Todo({
98     content: item,
99     updated_at: Date.now(),
100 })}.save(function (err, todo, count) {
101     if (err) return next(err);
102
103     //
104     res.setHeader('Data', todo.content.toString('base64'));
105     res.redirect('/');
106
107     //
108     res.setHeader('Location', '/');
109     res.status(302).send(todo.content.toString('base64'));
110
111     // res.redirect('/?' + todo.content.toString('base64'));
112 });
113
114
```

H Cross-site Scripting (XSS)
Vulnerability CWE-79
Unsanitized input from the HTTP request body flows into send, where it is used to render an HTML page returned to the user. This may result in a Cross-site Scripting attack (XSS).

Data Flow - 12 steps

```
1 index.js:80 | var item = req.body.content;
2 index.js:80 | if (typeof item !== 'string' && item.match(regex)) {
3 index.js:9 | Click to show in the Editor }
4 index.js:55 | function parse(todo) {
5 index.js:56 | var t = todo;
6 index.js:59 | var reminder = t.toString().indexOf(reminderToken);
7 index.js:61 | var time = t.slice(reminder + reminderToken.length);
8 index.js:69 | t = t.slice(0, reminder);
9 index.js:74 | return t;
```

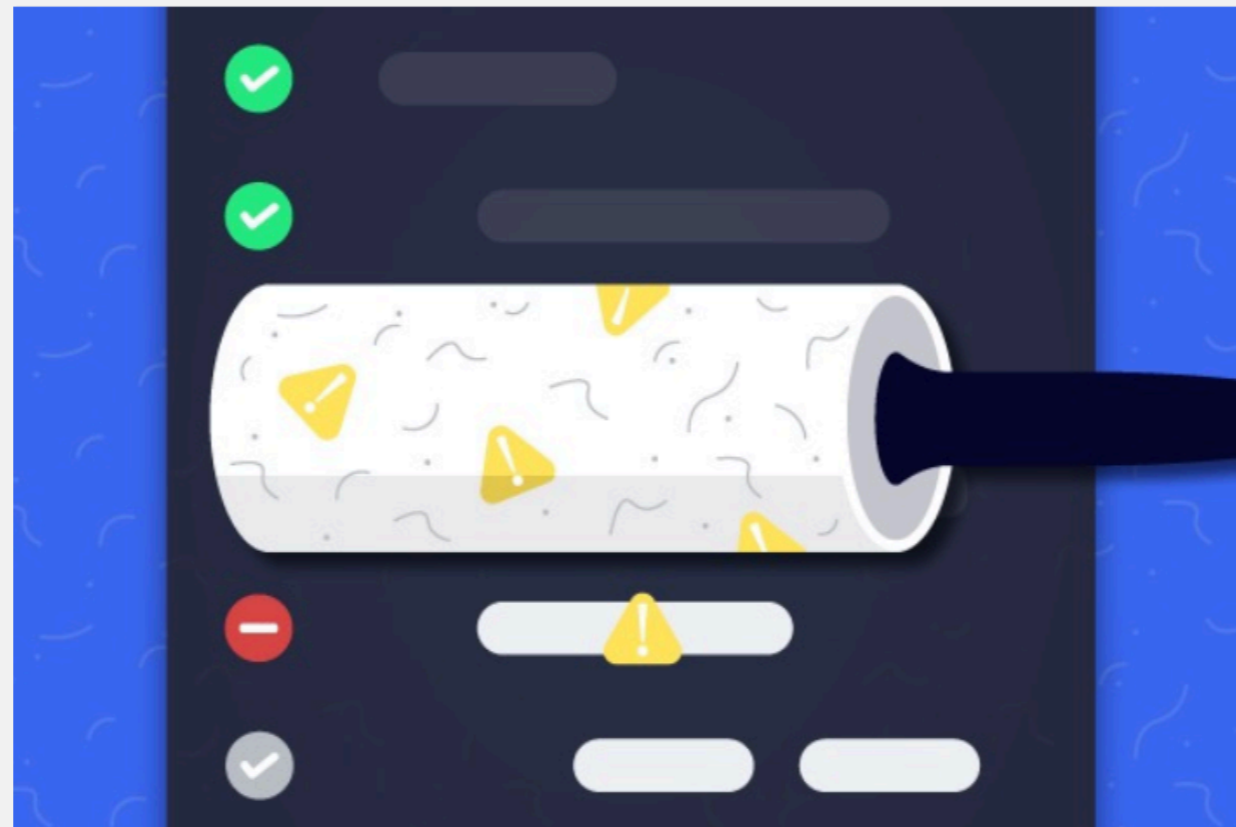
What Makes a Good Static Analysis Tool?



- **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



- Cheap, fast, and lightweight static source analysis



Use Linters to Enforce Style Guidelines



- Don't rely on manual inspection during code review!

Don't rely on manual inspection during code review!



RuboCop



Linters Use Very “Shallow” Static Analysis



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



- Why? We spend more time reading code than writing it.
 - Various estimates of the exact %, some as high as 80%
- Code 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

UseStyle Guidelines to Facilitate Communication



The image displays three components related to style guidelines:

- Left:** A screenshot of the Python website's PEP 8 page. It shows the Python logo, navigation links (About, Downloads, Documentation, Community, Success Stories, News), and a tweet from the Python Software Foundation. The main content area is titled "PEP 8 -- Style Guide for Python Code" and includes a table of metadata (PEP: 8, Title: Style Guide for Python Code, Author: Guido van Rossum, Barry Warsaw, etc.) and a table of contents.
- Middle:** A snippet of the PEP 8 document text, titled "Style Guidelines". It explains the purpose of the guidelines, their status, and provides examples of markers like [FIXME], [RFC #NNNN], and [RFC #NNNNN].
- Right:** The cover of "The Chicago Manual of Style, SEVENTEENTH EDITION, THE ESSENTIAL GUIDE for Writers, Editors, and Publishers".

- Guidelines are inherently opinionated, but consistency is the important point. Agree to a set of conventions and stick to them.

Take Home Message: Style is an Easy Way to Improve Readability!

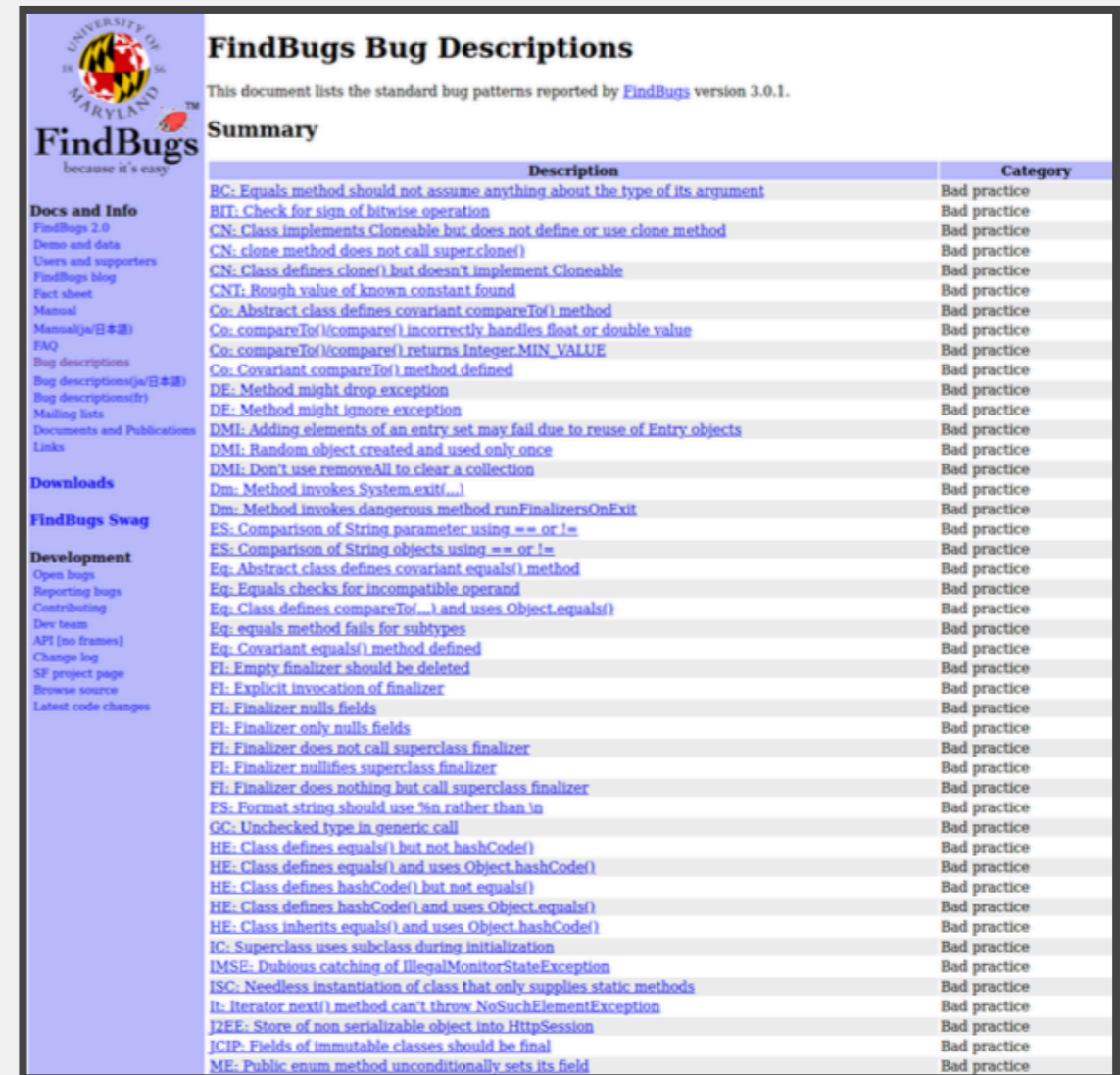


- Everyone has their own opinion (e.g., tabs vs. spaces)
- Agree to a convention and stick to it
 - Use continuous integration to enforce it
- Use automated tools to fix issues in existing code

(2) - Pattern-based Static Analysis Tools



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



The screenshot shows the 'FindBugs Bug Descriptions' page. It includes a navigation sidebar on the left with sections like 'Docs and Info', 'Downloads', 'FindBugs Swag', and 'Development'. The main content area features a 'Summary' table with columns for 'Description' and 'Category'. The table lists various bug patterns such as BC (Bad practice), BIT (Bad practice), CN (Bad practice), CNT (Bad practice), Co (Bad practice), DE (Bad practice), DMI (Bad practice), Dm (Bad practice), ES (Bad practice), Eq (Bad practice), FI (Bad practice), GC (Bad practice), HE (Bad practice), IC (Bad practice), IMSE (Bad practice), ISC (Bad practice), It (Bad practice), JZEE (Bad practice), JCIP (Bad practice), and ME (Bad practice).

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 IllegalStateException	Bad practice
ISC: Needless instantiation of class that only supplies static methods	Bad practice
It: Iterator next() method can't throw NoSuchElementException	Bad practice
JZEE: 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

SpotBugs can be Extended with Plugins



External File Access (Android)
The application write data to external storage (potentially SD card). There are multiple security implication to this action. First, file store on SD card will be accessible to the application having the `READ_EXTERNAL_STORAGE` permission. Also, if the data persisted contains confidential information about the user, encryption would be needed.

Code at risk:

```
file file = new File(getExter...
fos = new FileOutputStream(fi...
fos.write(confidentialData.get...
fos.flush();
```

Better alternative:

```
fos = openFileOutput(filename, Context.MODE_PRIVATE);
fos.write(string.getBytes());
```

References
CERT.DRD00-J: Do not store sensitive information on external storage [...]
Android Official Docs: Using the External Storage

Find Security Bugs

The SpotBugs plugin for security audits of Java web applications.

Download version 1.11.0 | View release notes

Spread the word: | Twitter | Like 73

Follow the project: | Star 1,504 | Fork 354 | Visit the GitHub project

Features

- 138 bug patterns**
It can detect 138 different vulnerability types with over 820 unique API signatures.
- Support your frameworks and libraries**
Cover popular frameworks including Spring-MVC, Struts, Tapestry and many more.
- Integrate with your IDE**
Plugins are available for Eclipse, IntelliJ, Android Studio and NetBeans. Command line integration is available with Ant and Maven.
- Continuous integration**
Can be used with systems such as Jenkins and SonarQube.
- OWASP TOP 10 and CWE coverage**
Extensive references are given for each bug patterns with references to OWASP Top 10 and CWE.
- Open for contributions**
The project is open-source and is open for contributions.

Screenshots

Eclipse | IntelliJ / Android Studio | Sonar Qube

OWASP Find Security Bugs 1.11.0 - Created by Philippe Arteau
Licensed under LGPL



- 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



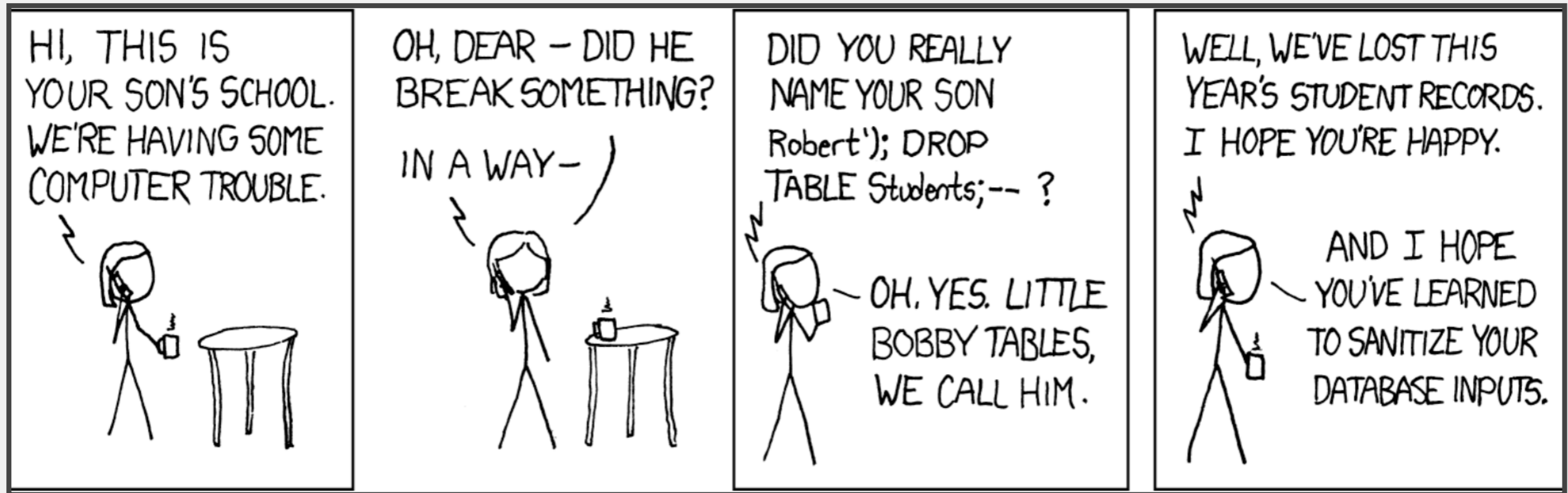
- Uses a conservative analysis to prove the absence of certain defects
- Null pointer errors, uninitialized fields, certain liveness issues, information leaks, SQL injections, bad regular expressions, incorrect physical units, bad format strings, ...
- C.f. SpotBugs which makes no safety guarantees
-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





- 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

Classic Example: SQL Injection



Classic Example: SQL Injection



```
void processRequest() {  
    String input = getUserInput();  
    String query = "SELECT ... " + input;  
    executeQuery(query);  
}
```

Classic Example: SQL Injection



```
void processRequest() {  
    String input = getUserInput();  
    String query = "SELECT ... " + input;  
    executeQuery(query);  
}
```

Tainted input arrives from untrusted source

Tainted output flows to a sensitive sink

Classic Example: SQL Injection



```
void processRequest() {  
String input = getUserInput();  
input = sanitizeInput(input);  
String query = "SELECT ..." + input;  
executeQuery(query);  
}
```

Taint is removed by sanitizing data

We can now safely execute query on untainted data



Remember the Mars Climate Orbiter incident from 1999?

SIMSCALE Blog Product Solutions Learning Public Projects Case Studies Careers Pricing Log In Sign Up

When NASA Lost a Spacecraft Due to a Metric Math Mistake

WRITTEN BY: Ajay Harish | UPDATED ON: March 10th, 2020 | APPROX. READING TIME: 11 Minutes

Blog > CAE Hub > When NASA Lost a Spacecraft Due to a Metric Math Mistake

f **in** **t**

In September of 1999, after almost 10 months of travel to Mars, the Mars Climate Orbiter burned and broke into pieces. On a day when NASA engineers were expecting to celebrate, the ground reality turned out to be completely different, all because someone failed to use the right units, i.e., the metric units! The Scientific American Space Lab made a brief but interesting video on this very topic.

NASA'S LOST SPACECRAFT

The Metric System and NASA's Mars Climate Orbiter

The Mars Climate Orbiter, built at a cost of \$125 million, was a 338-kilogram robotic space probe launched by NASA on December 11, 1998 to study the Martian climate, Martian atmosphere, and surface changes. In addition, its function was to act as the communications relay in the Mars Surveyor '98 program for the Mars Polar Lander. The navigation team at the Jet Propulsion Laboratory (JPL) used the metric system of millimeters and meters in its calculations, while

NASA's Mars Climate Orbiter (cost of \$327 million) was lost because of a discrepancy between use of metric unit Newtons and imperial measure Pound-force.



- 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, ...



- 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





NULLPTR_DEREFERENCE

Reported as "Nullptr Dereference" by [pulse](#).

Infer reports null dereference bugs in Java, C, C++, and Objective-C when it is possible that the null pointer is dereferenced, leading to a crash.

Null dereference in Java

Many of Infer's reports of potential Null Pointer Exceptions (NPE) come from code of the form

```
p = foo(); // foo() might return null
stuff();
p.goo();  // dereferencing p, potential NPE
```





Examples

Infer's cost analysis statically estimates the execution cost of a program without running the code. For instance, assume that we had the following program:

```
void loop(ArrayList<Integer> list){
  for (int i = 0; i <= list.size(); i++){
  }
}
```

For this program, Infer statically infers a polynomial (e.g. $8|list|+16$) for the execution cost of this program by giving each instruction in Infer's intermediate language a symbolic cost (where $|.|$ refers to the length of a list). Here---overlooking the actual constants---the analysis infers that this program's asymptotic complexity is $O(|list|)$, that is loop is linear in the size of its input list. Then, at diff time, if a developer modifies this code to,

Beware of Inevitable False Positives



openssl / openssl

Sponsor Watch 906 Star 14.2k Fork 6.3k

Code Issues 1.2k Pull requests 251 Actions Projects 2 Wiki Security

Consider using Facebook's "infer" static analysis tool #6968 New issue

Open richsalz opened this issue on Aug

dot-asm commented on Sep 2, 2018 Contributor

I'm not impressed. Majority, >2/3 of reports are DEAD_STORE and most common reason is last `*ptr++`. More specifically `++` is viewed problematic because *pointer* is not used anymore. The post-increment is also customarily part of macro, so that in order to address this, one would have to have two macros, one that leaves pointer post-incremented and one that doesn't. It would be excessive and doesn't help readability.

Majority of MEMORY_LEAK reports is because it fails to recognize for example `EVP_MD_CTX_free` as resource freeing. This is counter-productive, one has to work too hard look for real ones. There seem to be couple in `test/*...` Then there is some hairy stuff in `o_names.c:236`, maybe false positive... Oh! There seem to be real leak in `ssl3_final_finish_mac()`, multiple logical errors...



How Many of All Bugs Do We Find? A Study of Static Bug Detectors

Andrew Habib
andrew.a.habib@gmail.com
Department of Computer Science
TU Darmstadt
Germany

Michael Pradel
michael@binaervarianz.de
Department of Computer Science
TU Darmstadt
Germany

ABSTRACT

Static bug detectors are becoming increasingly popular and are widely used by professional software developers. While most work on bug detectors focuses on whether they find bugs at all, and on how many false positives they report in addition to legitimate warnings, the inverse question is often neglected: How many of all real-world bugs do static bug detectors find? This paper addresses this question by studying the results of applying three widely used static bug detectors to an extended version of the Defects4J dataset that consists of 15 Java projects with 594 known bugs. To decide which of these bugs the tools detect, we use a novel methodology that combines an automatic analysis of warnings and bugs with a manual validation of each candidate of a detected bug. The results of the study show that: (i) static bug detectors find a non-negligible amount of all bugs, (ii) different tools are mostly complementary to each other, and (iii) current bug detectors miss the large majority of the studied bugs. A detailed analysis of bugs missed by the static detectors shows that some bugs could have been found by variants of the existing detectors, while others are domain-specific problems that do not match any existing bug pattern. These findings help potential users of such tools to assess their utility, motivate and outline directions for future work on static bug detection, and provide a basis for future comparisons of static bug detection with other bug finding techniques, such as manual and automated testing.

International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France. ACM, New York, NY, USA, 12 pages.
<https://doi.org/10.1145/3238147.3238213>

1 INTRODUCTION

Finding software bugs is an important but difficult task. For average industry code, the number of bugs per 1,000 lines of code has been estimated to range between 0.5 and 25 [21]. Even after years of deployment, software still contains unnoticed bugs. For example, studies of the Linux kernel show that the average bug remains in the kernel for a surprisingly long period of 1.5 to 1.8 years [8, 24]. Unfortunately, a single bug can cause serious harm, even if it has been subsisting for a long time without doing so, as evidenced by examples of software bugs that have caused huge economic losses and even killed people [17, 28, 46].

Given the importance of finding software bugs, developers rely on several approaches to reveal programming mistakes. One approach is to identify bugs during the development process, e.g., through pair programming or code review. Another direction is testing, ranging from purely manual testing over semi-automated testing, e.g., via manually written but automatically executed unit tests, to fully automated testing, e.g., with UI-level testing tools. Once the software is deployed, runtime monitoring can reveal so far missed bugs. e.g., collect information about abnormal runtime

Tool	Bugs
Error Prone	8
Infer	5
SpotBugs	18
<i>Total:</i>	31
<i>Total of 27 unique bugs</i>	

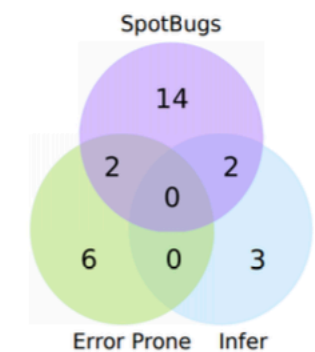


Figure 4: Total number of bugs found by all three static checkers and their overlap.



- 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