Course Notes Set 4:

Software Process Maturity

Computer Science and Software Engineering
Auburn University

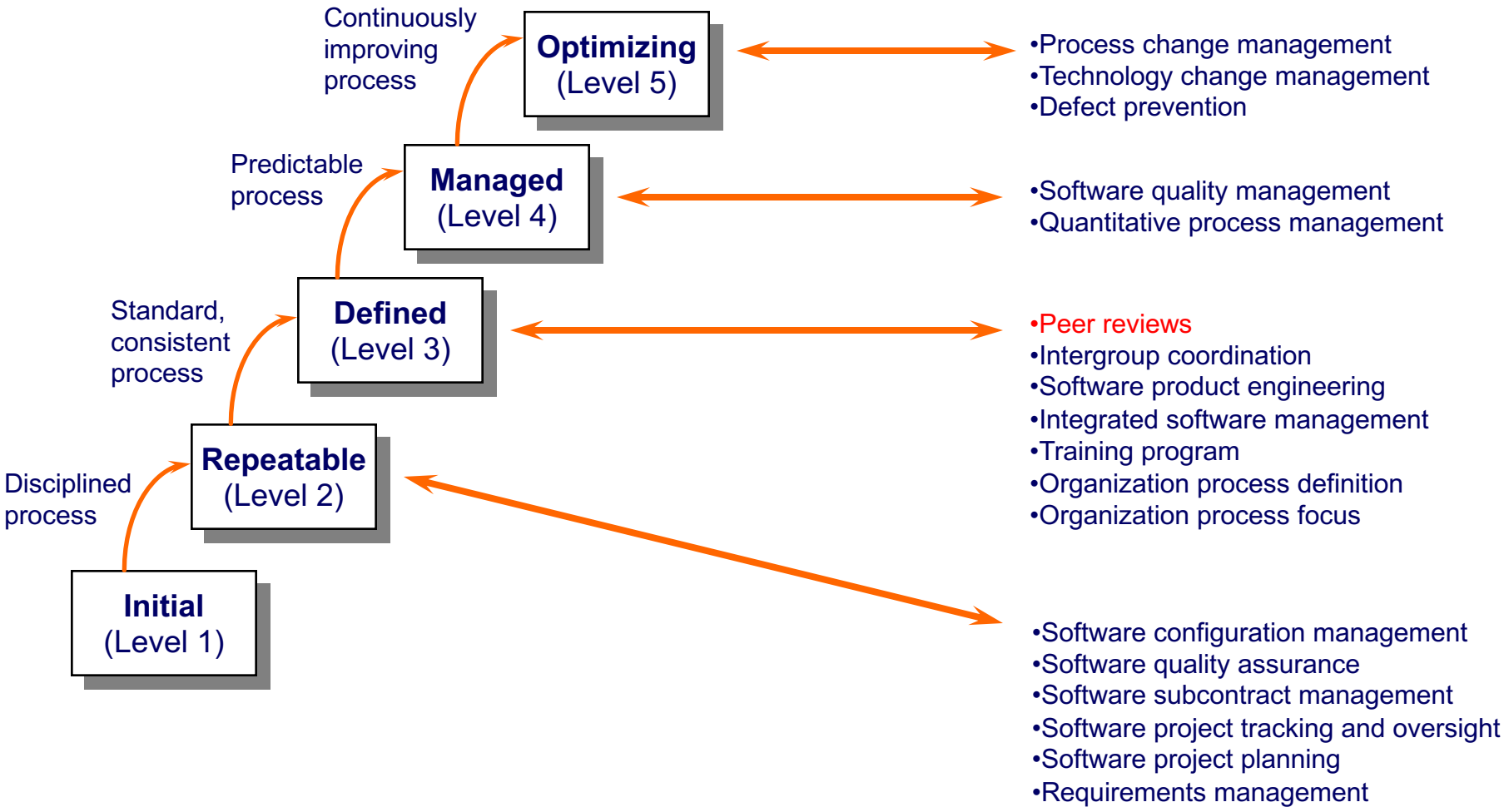
Software Process Maturity

- A measure of the long-term effectiveness of an organization's software engineering practices.
- Being rated at a certain level of maturity can be a contractual obligation for the software developer, required by the client.
- Several approaches to describing/measuring/promoting process maturity:
  - Capability Maturity Model (CMM) - SEI
  - Baseline Practices Guide - ISO/IEC
  - Trillium - Northern Telecom/BNR

Capability Maturity Model

- Began in 1986 as the SEI and MITRE responded to the DoD's demands for better software.
- First brief description released in 1987 by Watts Humphrey.
- Formally evolved into the CMM, authored by Mark Paulk in 1992.
- Principle ideas:
  - Describes the key elements of an effective software process.
  - Describes an evolutionary improvement path from an ad hoc, immature process to a disciplined, mature process.
  - The CMM is composed of five *maturity levels*; all but the first (Level 1) are, in turn, composed of *key process areas* (KPA). Each KPA is organized into five sections called *common features* which specify *key practices*. The key practices, when collectively addressed, accomplish the goals of the given KPA.
  - An organization is graded according to the presence of key practices that support KPAs.

CMM Maturity Levels and KPAs



[Adapted from Technical Report SEI-93-TR-24]

Characteristics of Immaturity

- Software process improvised during the course of a project.
- Even if process is specified, it is not rigorously followed or enforced.
- Reactionary, focus on solving immediate crises.
- Hard deadlines often mean a compromise in functionality and/or quality.
- No objective basis for judging product quality or for solving process problems.
- Quality is difficult if not impossible to predict.

Characteristics of Maturity

- Able to manage software development and maintenance organization/project wide.
- There is a prescribed, mandated, and enforced process.
- Process is consistent with the way that work actually gets done.
- Process is updated and improved as necessary.
- Roles and responsibilities within the process are clear.
- Quality is measured and monitored, and an objective basis for judgment exists.
- The necessary infrastructure for supporting the process exists.
- Workers see the value in the process.

A Survey of High Maturity Organizations

- No organization type or structure seems to be correlated with maturity.
- Most have multiple quality improvement initiatives.
- Typically use incremental or evolutionary process models.
- Most measure customer/user satisfaction and have meaningful interaction with them.
- Most use cost models (functionality, schedule => cost)
- Typically incorporate risk management into their process.
- Typically have an independent SQA group and embed SQA into the process.
- Typically use Internet/intranet to deploy process assets.
- Most use a consistent, though not formal, process notation.
- Most have required training in "people issues" - interpersonal skills, team building.
- Typically use both inspections and walkthroughs

[From "Practices of High Maturity Organizations: The 1999 Survey" by Paulk, Goldenson, and White, December 1999.]

Greater Maturity Can Bear Fruit

- SE division of Hughes aircraft spent @$500K over a three year period for assessment and improvement programs. By the end of the three year period, assessed at CMM Level 3. Estimated savings of @$2M annually as a result (less overtime, less rework, greater productivity, etc.)
- Equipment Division of Raytheon rise to CMM Level 3, at an estimated cost of @$580K resulted in 2-fold increase in productivity along with savings of @$15.8M in rework costs.
- Motorola GED (CMM Level 4) documented significant
  - reduction in cycle time
  - reduction in defect rates
  - increase in productivity

Capability Maturity Model Integration - CMMI

- So far CMM – for software (SW-CMM)
- Issues in a system --> more than software
  - e.g. system engineering, electronics, manufacturing

- Many CMMI models – different emphasis

- Based on
  - Capability Maturity Model for Software (SW-CMM)
  - System Engineering Capability Model (SECM)
  - Integrated Product Development Capability Maturity Model (IPD-CMM)

- http://www.sei.cmu.edu/cmmi

Improvement over SW-CMM

- More explicit linkage between management and engineering
- More coverage in life cycle and engineering activities
- Incorporation of best practices from additional areas, e.g., management, measurement
- Additional organizational functions
- More compliance with relevant ISO standards
- More proven and robust high maturity practices

Why CMMI?

- Specific CMM models for different disciplines
- Expensive for various adoptions, trainings, and management styles across an organization
- Integration !!!

- Two different representations
  - Staged and Continuous

- Easier to migrate from SW-CMM to CMMI using staged representation

Maturity Levels for CMMI

- Level 1 - Initial
- Level 2 - Managed
- Level 3 - Defined
- Level 4 - Quantitatively Managed
- Level 5 - Optimizing

References

[1] M. Diaz and J. Sligo, "How software process improvement helped Motorola," in IEEE Software, vol. 14, no. 5, pp. 75-81, Sept.-Oct. 1997, doi: 10.1109/52.605934.

[2] Glazer, Hillel., Dalton, Jeff., Anderson, David., Konrad, Michael., & Shrum, Sandra. (2008). *CMMI or Agile: Why Not Embrace Both!* (CMU/SEI-2008-TN-003). Retrieved February 10, 2022, from the Software Engineering Institute, Carnegie Mellon University website: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8533

https://twitter.com/warren_craddock/status/1579532951624175616

COMP 5710/6710: Software Quality Assurance

Akond Rahman, PhD
Department of Computer Science and Software Engineering (CSSE)
Shelby Center 3104
akond@auburn.edu
https://akondrahman.github.io

AUBURN

About Me: Academia

Assistant Professor

Assistant Professor

PhD

MSc

BSc

AUBURN

About Me: Industry

IBM Research

AUBURN

Announcements

- Project team formation: https://forms.office.com/r/rdEtmnX7B4
- Deadline: Oct 01, 2025

- Are you interested in extra credit?
  Full participation in the extra credit will result in 5 points for the midterm exam. You have to register and
  participate in the Cyber Fire Puzzle competition. Once verified by Dr. Tauritz, I will add 5 points to the midterm exam.

AUBURN

Office Hours

- Akond Rahman, Shelby 3137, Wednesday, 11 AM – 11:30 AM CST
- TA, Md. Jahidul Arafat

AUBURN

Why Software Quality Assurance (SQA)?

Software Assurance and Software Safety

Software Assurance is defined as the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in an intended manner.

The objective of NASA Software Assurance and Software Safety is to ensure that the processes, procedures and products used to produce and sustain NASA software conform to all requirements and standards specified to govern those processes, procedures and products.

THE WHITE HOUSE

Administration Priorities COVID Plan Briefing Room Español MENU

BRIEFING ROOM

Executive Order on Improving the Nation's Cybersecurity

MAY 12, 2021 • PRESIDENTIAL ACTIONS

Application Security | 5 MIN READ | ARTICLE

Testing & Automation Pay Off for NSA's DevSecOps Project

Communication with stakeholders, extensive testing, and robust automation pays dividends for military intelligence agency, one of several presenters at GitLab's virtual Commit conference.

Robert Lemos
Contributing Writer, Dark Reading

August 31, 2020

About the Class

Syllabus: Visit CANVAS
Questions: CANVAS and Office Hours
Schedule: https://github.com/paser-group/continuous-secsoft/tree/master/fall25-sqa
Focus:

- Hands-on experience in application and construction of tools
- Instructor-led authentic learning-based workshops

AUBURN

What Can You Expect?

After you are done with this class:

1. Knowledge on software quality assurance activities, such as testing and security tools
2. Application and construction of program analysis tools
3. State-of-the-art software engineering techniques

Software Needed

1. Git
2. Python (We will use basic Python)
3. Pip
4. Docker
5. Other libraries as class progresses

AUBURN

COMP 5710/6710: Software Quality Assurance

Akond Rahman, PhD
Department of Computer Science and Software Engineering (CSSE)
akond@auburn.edu
https://akondrahman.github.io

AUBURN

SQA: What is Software Quality?

Quality – developed product meets its specification

Quality Management – ensuring that required level of product quality is achieved

- Defining procedures and standards
- Applying procedures and standards to the product and process
- Checking that procedures are followed
- Collecting and analyzing various quality data

AUBURN

SQA: What is Software Quality Assurance?

Establishment of network of organizational
procedures and standards leading to high-quality software

AUBURN

SQA: Metrics

- Measurement

  is the act of obtaining a measure

- Measure

  provides a quantitative indication of the size of some product or process attribute,
  E.g., Number of errors

- Metric

  is a quantitative measure of the degree to which a system, component, or process possesses a given attribute
  e.g., Number of errors found per person hours expended

AUBURN

SQA: Importance of Metrics

- Determine quality of piece of software or documentation
- Determine the quality work of people such
  software engineers, programmers, database admin, and managers
- Improve quality of a product/project/ process

AUBURN

SQA: Categories of Metrics

Code quality
Programmer productivity
Software engineer productivity
    Requirements,
    design,
    testing
    and all other tasks done by software engineers
Software
    Maintainability
    Usability
    And all other quality factors
Management
    Cost estimation
    Schedule estimation, Duration, time
    Staffing

AUBURN

SQA: Bugs and Vulnerabilities

1. Bug/Defect: An imperfection that needs to be repaired of replaced
2. Vulnerabilities: A bug/defect that violates confidentiality, integrity, or availability.

AUBURN

SQA: What Do Professionals Do?

https://youtu.be/ue0dkFd748g?t=155

AUBURN

COMP 5710/6710: Software Quality Assurance

Akond Rahman, PhD
Department of Computer Science and Software Engineering (CSSE)
akond@auburn.edu
https://akondrahman.github.io

AUBURN

Software Development Lifecycle Model

- A framework that describes the activities performed at each stage of a software development project.

AUBURN

Waterfall Strengths

- Easy to understand, easy to use
- Provides structure to inexperienced staff
- Milestones are well understood
- Sets requirements stability
- Good for management control (plan, staff, track)
- Works well when quality is more important than cost or schedule

AUBURN

Waterfall Limitations

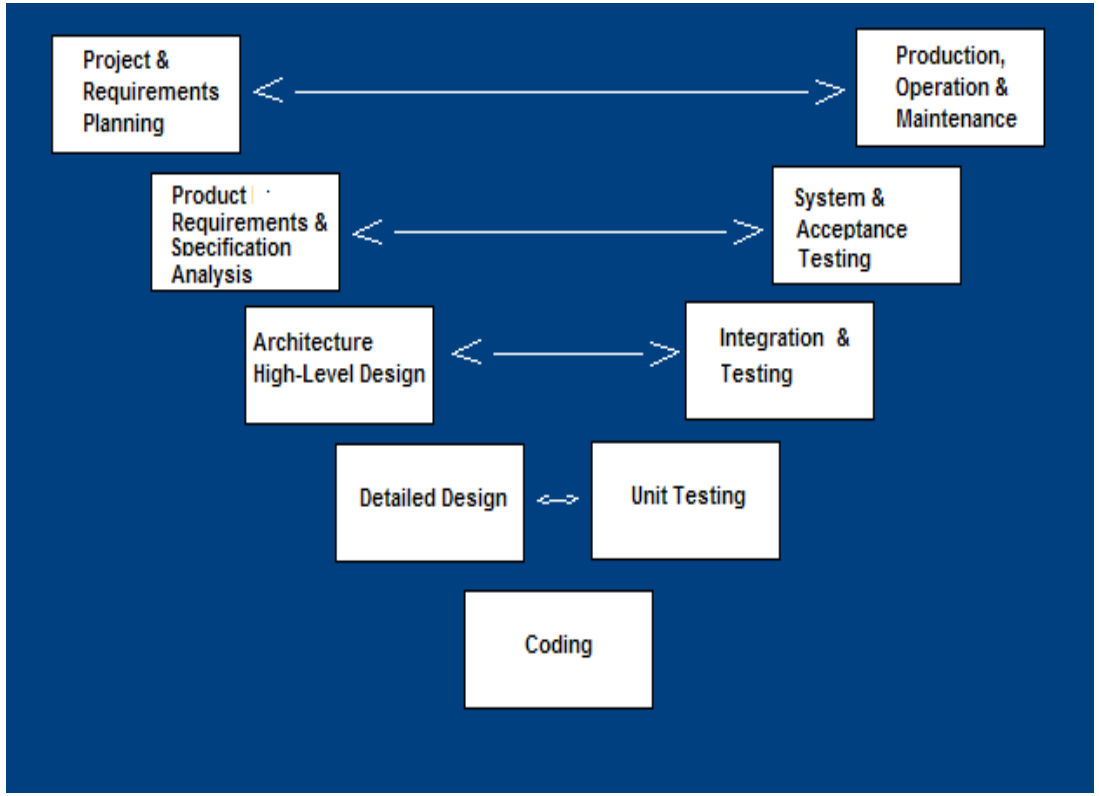
- All requirements must be known upfront
- Deliverables created for each phase are considered frozen – inhibits flexibility
- Can give a false impression of progress
- Does not reflect problem-solving nature of software development – iterations of phases
- Integration is one big bang at the end
- Little opportunity for customer to preview the system (until it may be too late)

Waterfall – When to Use

- Requirements are very well known
- Product definition is stable
- Technology is understood
- New version of an existing product
- Porting an existing product to a new platform.

AUBURN

V-Shaped SDLC Model



- A variant of the Waterfall that emphasizes the verification and validation of the product.
- Testing of the product is planned in parallel with a corresponding phase of development

AUBURN

Steps in V-Shaped SDLC Model

- Project and Requirements Planning – allocate resources
- Product Requirements and Specification Analysis – complete specification of the software system
- Architecture or High-Level Design – defines how software functions fulfill the design
- Detailed Design – develop algorithms for each architectural component
- Production, operation and maintenance – provide for enhancement and corrections
- System and acceptance testing – check the entire software system in its environment
- Integration and Testing – check that modules interconnect correctly
- Unit testing – check that each module acts as expected
- Coding – transform algorithms into software

AUBURN

Strengths of V-Shaped SDLC Model

- Emphasize planning for verification and validation of the product in early stages of product development
- Each deliverable must be testable
- Project management can track progress by milestones
- Easy to use

AUBURN

Limitations of V-Shaped SDLC Model

- Does not easily handle concurrent events
- Does not handle iterations or phases
- Does not easily handle dynamic changes in requirements
- Does not contain risk analysis activities

AUBURN

When to Use V-Shaped SDLC Model

- Excellent choice for systems requiring high reliability – hospital patient control applications
- All requirements are known up-front
- When it can be modified to handle changing requirements beyond analysis phase
- Solution and technology are known

AUBURN

Structured Evolutionary Prototyping Model

- Developers build a prototype during the requirements phase
- Prototype is evaluated by end users
- Users give corrective feedback
- Developers further refine the prototype
- When the user is satisfied, the prototype code is brought up to the standards needed for a final product.

AUBURN

Steps of Structured Evolutionary Prototyping Model

- A preliminary project plan is developed
- A partial high-level paper model is created
- The model is source for a partial requirements specification
- A prototype is built with basic and critical attributes
- The designer builds
  - the database
  - user interface
  - algorithmic functions
- The designer demonstrates the prototype, the user evaluates for problems and suggests improvements.
- This loop continues until the user is satisfied

AUBURN

Strengths of Structured Evolutionary Prototyping Model

- Customers can “see” the system requirements as they are being gathered
- Developers learn from customers
- A more accurate end product
- Unexpected requirements accommodated
- Allows for flexible design and development
- Steady, visible signs of progress produced
- Interaction with the prototype stimulates awareness of additional needed functionality

AUBURN

Limitations of Structured Evolutionary Prototyping Model

- Tendency to abandon structured program development for “code-and-fix” development
- Bad reputation for “quick-and-dirty” methods
- Overall maintainability may be overlooked
- The customer may want the prototype delivered.
- Process may continue forever (scope creep)

When to Use Structured Evolutionary Prototyping Model

- Requirements are unstable or have to be clarified
- As the requirements clarification stage of a waterfall model
- Develop user interfaces
- Short-lived demonstrations
- New, original development
- With the analysis and design portions of object-oriented development

AUBURN

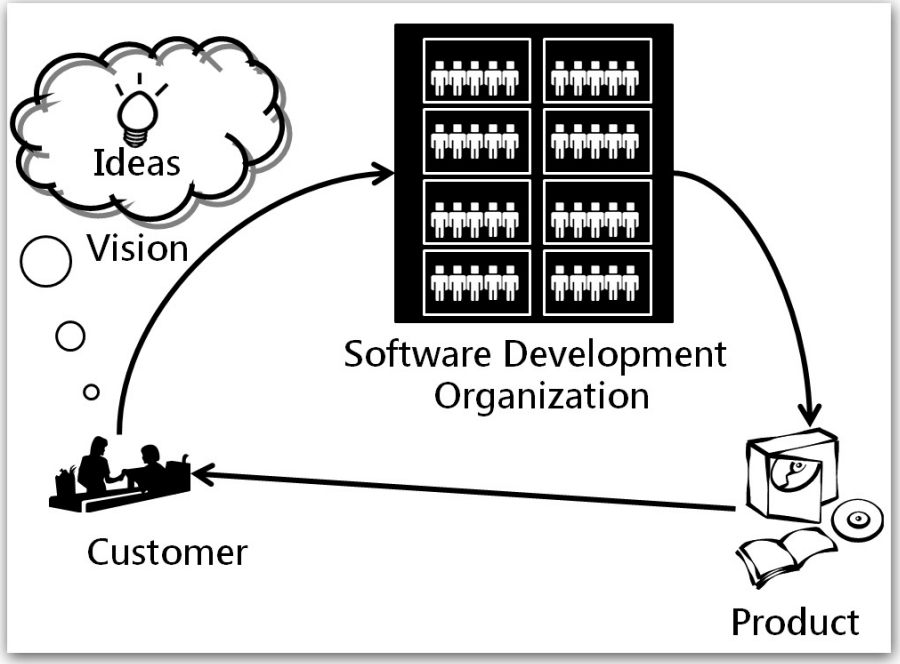
Course Notes Set 2:

Software Process - Scrum

Computer Science and Software Engineering
Auburn University

Software Production

- Turing a vision into a software product. When deployed in the intended environment, the vision is realized.



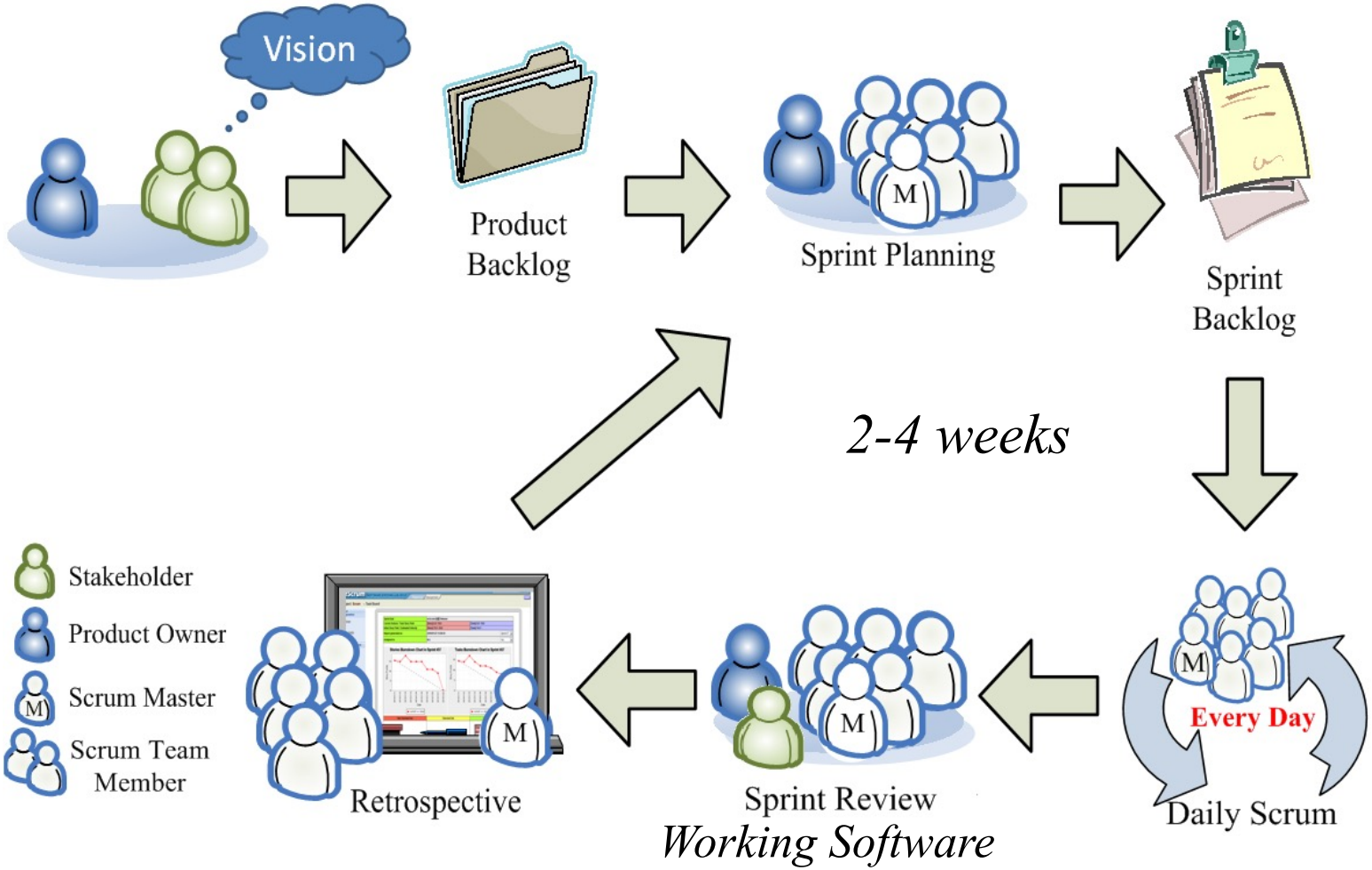
Software Process Models

- The way it often happens...
  - Build-and-fix, a.k.a "None"
- Some models of "how it should be"...
  - Waterfall*
  - Incremental
  - Spiral
  - Cleanroom
- Some models of "how it's getting done"…
  - Scrum*
  - XP
- Some helpful relatives…
  - Prototyping
  - Formal Methods

Build the right thing; build the thing right

- Client needs to have many opportunities to check whether they have the right thing.
  - Boehm : "IKIWISI" (I'll know it when I see it)
- Client has the right to *pivot* to get the best value
- Developers have the responsibility
  - to build the thing right through *Working Software*
  - to adapt to change
  - build quality in

Scrum: how it's getting done



Product Owner

- Collects and organizes requirements into *user stories* with *product backlog*
  - Release planning
- Prioritizes and scope user stories
  - Select user stories based on priority and team velocity
- Clarifies requirements
  - on and off sprint planning
- Accepts/rejects the implemented user stories

Developers

- Delivers *potential shippable product* at the end of a sprint
- *Cross-functional*: all the talents to complete all work *inside* the team
- Self-organize and self-manage
- Review work results with PO

Scrum Master

"Scrum is simple but hard."

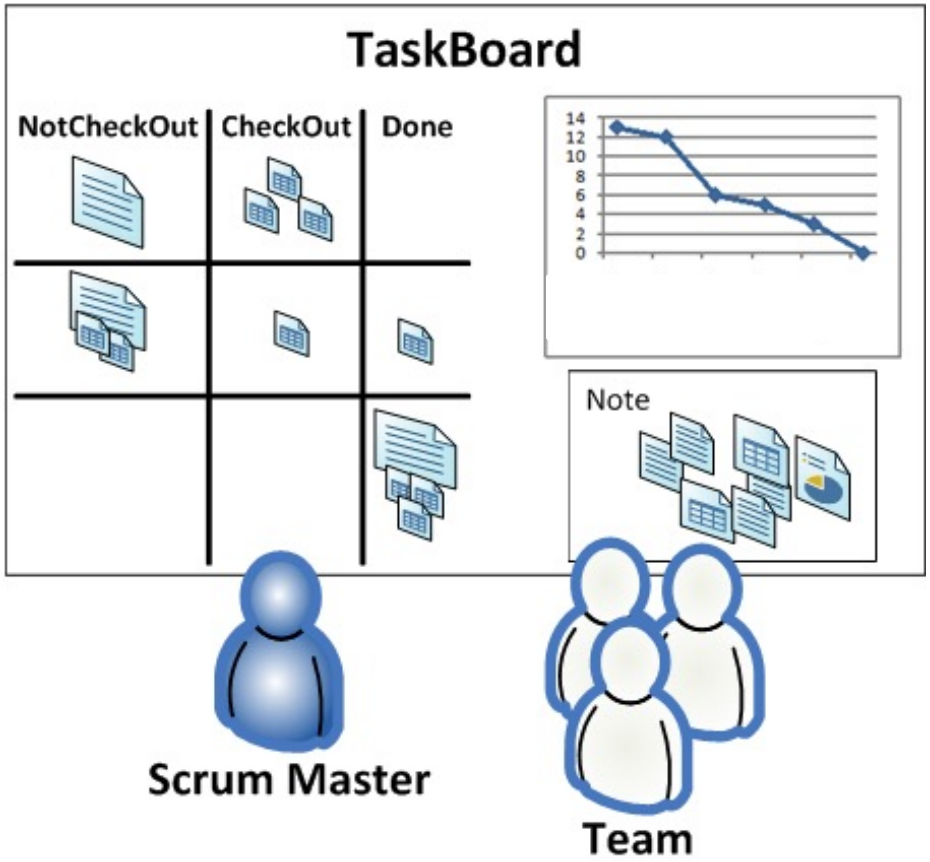
- Coaching Scrum
- Help team members focus on their tasks at hand
- Facilitate meetings but does not make decisions for the team or PO
- Help the team uncovers hidden difficulties
- Help the team acquire needed resources

Team Activities (1/2)

- In Sprint Planning:
  - *Estimate* user story size
  - Divide a user story into *smaller tasks* and estimate time
- In Daily Standup:
  - What did I do yesterday?
    - Update story/task stage, re-estimate remain hours
  - What will I do today? (Claim new tasks)
  - Any difficulties?

Daily Scrum

Daily Scrum



Team Activities (2/2)

- In Sprint Demo (Review)
  - Demonstrate completed user stories as primarily as working software
  - Discussion more important than demo
- In Sprint retrospective
  - What did we do well?
  - What can be improved?
  - What actions to take?
- … and develop product, of course!

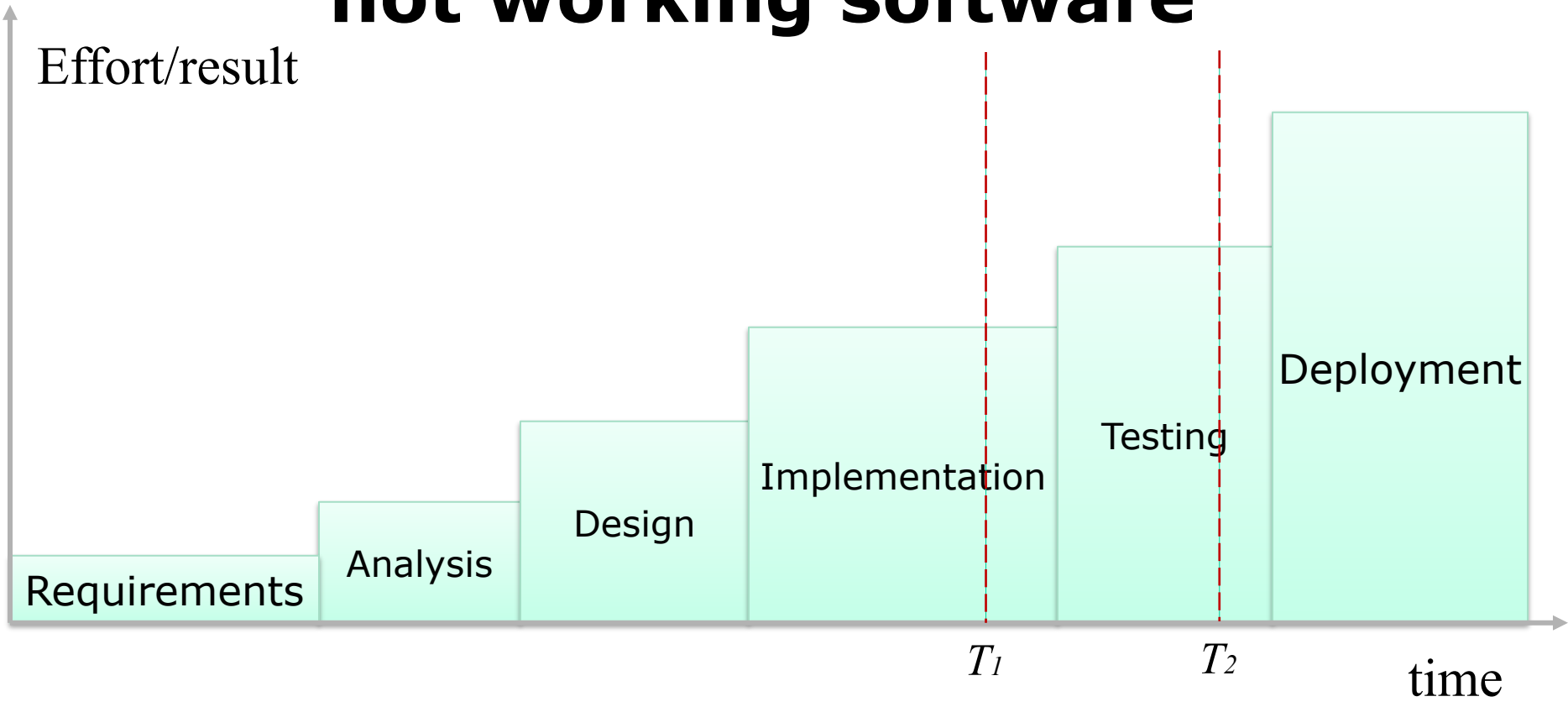
Scrum Artifacts (1)

- User stories (Product backlog items): Unit of work for the team that are deliverable and verifiable
- Product backlog: where user stories and organized and maintained
- Sprint backlog: the selected user stories and their constituent tasks

Scrum Artifacts (2)

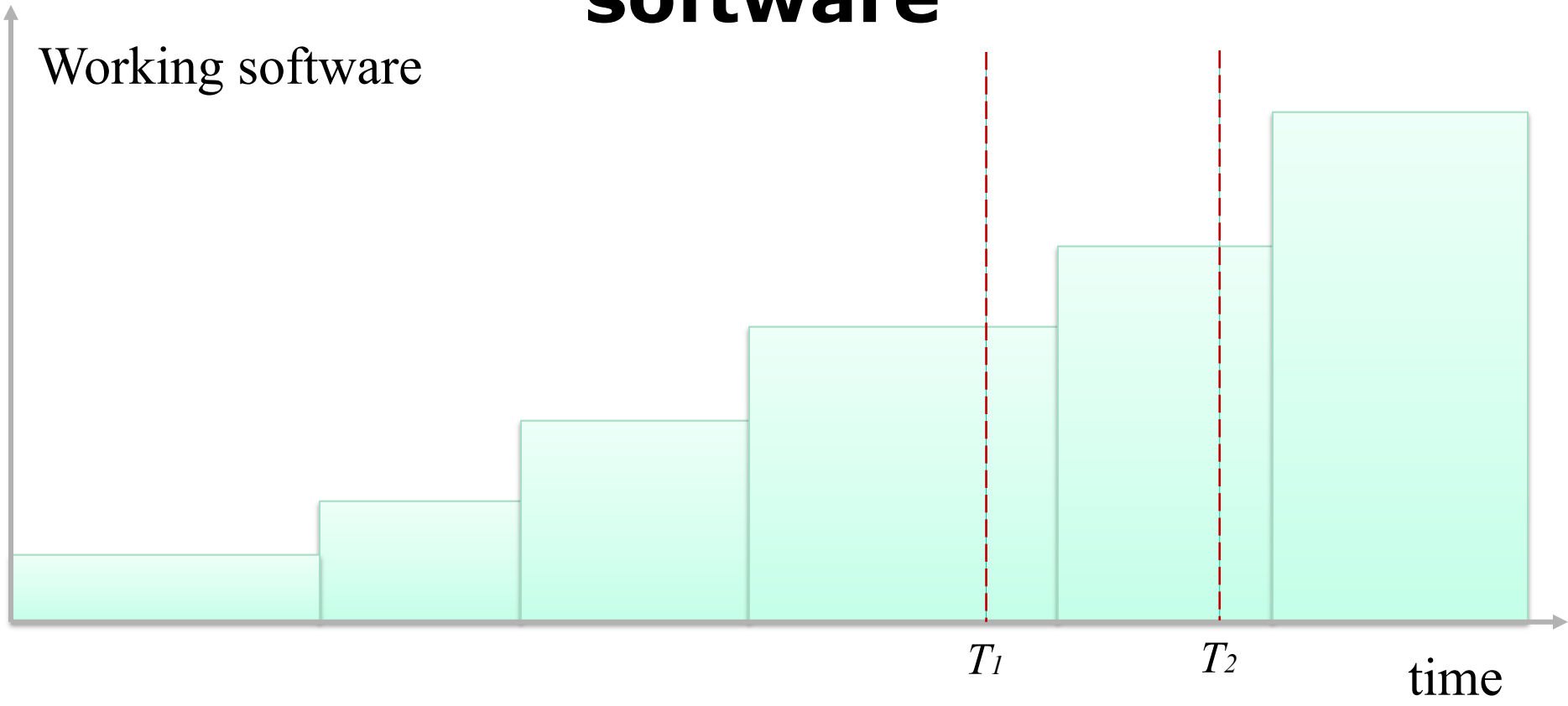
- Burndown charts: how much have we accomplished so far?
  - User story points for PO and team
  - Task hours for team
- Task Board: stories/tasks by stages + burndown charts

Waterfall: documents, code but not working software



What if we get it wrong?

Scrum: delivering working software



Iterative towards better capability

- Defintion of Done:
  - Begining with requiring tests
  - Adding continuous integration
  - Adding static analysis
  - Moving to pair programming

Agile manifesto

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

Mob programming/Strong-styled pair programming

- One person be the driver in charge of keyboard and mouse
- The other person be the navigator to say what to do
- Pair programming is always a possible choice, especially if someone runs into difficulty
- Mob programming can still be a choice especially on tasks that affect the whole team

References

- Ken Schwaber & Jeff Sutherland, *The Scrum Guide: the Definitive Guide to Scrum: The Rules of the Game. Nov 2020.* https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf#zoom=100

COMP 5710/6710: Software Quality Assurance

Akond Rahman, PhD
Department of Computer Science and Software Engineering (CSSE)
akond@auburn.edu
https://akondrahman.github.io

AUBURN

What is a requirement?

- It is about WHAT not HOW
- Nothing can be said obvious
- Requirements are the descriptions of the services provided by a system and its operational constraints
- It may range from a high level abstract statement to a detailed mathematical specification
- It may be as complex as a 500 pages of description
- It may serve as the basis for a bid for a contract or the basis for the contract itself

AUBURN

What is requirements engineering?

- It is the process of discovering, analyzing, documenting and validating the requirements of the system
- Each software development process goes through the phase of requirements engineering

AUBURN

Why requirements

What are the advantages of a complete set of documented requirements?

- Ensures the user (not the developer) drives system functionalities
- Helps avoiding confusion and arguments
- Helps minimizing the changes

Changes in requirements are expensive. Changing the requirements costs:

- 3 x as much during the design phase
- 5-10 x as much during implementation
- 10-100 x as much after release

AUBURN

Types of requirements

- Functional requirements
  - What the system should do or not in reaction to particular situations
- Non-functional requirements
  - Constraints on the services or functions offered by the system
  - Examples: Timing constraints, constraints on the development process (CASE, language, development method…), standards etc.

Non-functional requirements

- They can be further categorized into:
  - Product requirements
    - Product behavior
    - Ex: Timing, performance, memory, reliability, portability, usability
  - Organizational requirements
    - Policies and procedures in the customer's and developer's organizations
    - Example: Process requirements, implementation requirements, delivery requirements
  - External requirements
    - Factors externals to the system and the development process
    - Example: Interoperability, legislation, ethics

AUBURN

Non-functional requirements

- It is important to be able to test/verify/check non-functional requirements

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

AUBURN

Requirements validation

- Concerned with demonstrating that the requirements define the system
  that the customer really wants
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of
    fixing an implementation error
- Prototyping is an important technique of requirements validation

AUBURN

Validation vs Verification

Validation	Verification
The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements	The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase
validation ensures that the software product meets the needs of all the stakeholders	verification ensures that the output of each phase of the software development process effectively carry out what its corresponding input artifact specifies
you built the right thing	you built it right
The actual product / software.	Plans, Requirement Specs, Design Specs, Code, Test Cases

AUBURN

Requirements checking

Validity. Does the system provide the functions which best support the customer's needs?
Consistency. Are there any requirements conflicts?
Completeness. Are all functions required by the customer included?
Realism. Can the requirements be implemented given available budget and technology

AUBURN

Requirements evolution

Requirements **always evolve** as a better understanding of user needs is developed
and as the organization's objectives change
It is essential to **plan for change** in the requirements as the system is being developed and used

AUBURN

Test-driven development for software requirements

Tools to implement: unittest

Security requirements

- A definition of information security with a clear statement of management's intentions
- An explanation of specific security requirements including:
  - Compliance with legislative and contractual requirements
  - Security education, virus prevention and detection, and business continuity planning
  - A definition of general and specific roles and responsibilities for the various aspects of information security program in business
  - an explanation of the requirement and process for reporting suspected security incidents, and
  - the process, including roles and responsibilities, for maintaining the policy document.

AUBURN

Structure of Security requirements

- Security requirements have a well-defined structure that consists of the following components:
  - ***Basic security requirements section.***
  - ***Derived security requirements section.***

AUBURN

Why Security requirements

- Internet
- Systems often control money
- Systems nearly always contain data (much of it personal)
- You are legally required often to keep your system secure
- You could get sued

AUBURN

Concepts related to Security requirements

- Broken down into 3 main issues
  - Confidentiality
  - Integrity
  - Availability

AUBURN

Specifying Security requirements

- Ideally kept as open as possible to allow for
  - Upgrading of encryption algorithms and protocols
- Security policy
  - Shredding documents
  - Secure disposal
  - Password reset protocols
  - Security training
  - Security audits
- Standards compliance
  - Payment Card Industry Data Security Standard

AUBURN

Example Security Requirement #1

Basic Security Requirement:

Ensure that organizational personnel are adequately trained to carry out their assigned information security-related duties and responsibilities.

Meeting the Requirement:

- Basic security awareness training to new employees.
- Security awareness training to users when information system changes.
- Annual security awareness refresher training.

AUBURN

Example Security Requirement #1

Basic Security Requirement:

Ensure that organizational personnel are adequately trained to carry out their assigned information security-related duties and responsibilities.

Meeting the Requirement:

- Security awareness and training policy.
- Security awareness training materials.
- Security plan; training records; other relevant documents or records.
- Personnel with responsibilities for security awareness training.

AUBURN

Example Security Requirement #2

Basic Security Requirement:

Establish and maintain baseline configurations and inventories of organizational information systems (including hardware, software, firmware, and documentation) throughout the respective system development life cycles.

Meeting the Requirements:

- Develops, documents and maintains a current baseline configuration of the information system
- Configuration control in place.

AUBURN

Example Security Requirement #2

Basic Security Requirement:

Establish and maintain baseline configurations and inventories of organizational information systems (including hardware, software, firmware, and documentation) throughout the respective system development life cycles.

Meeting the Requirements:

- Configuration management policy; procedures and plan.
- Documentation for Enterprise architecture or information system design.
- Information system configuration settings and associated documentation.
- Change control records.
- Personnel with configuration management responsibilities.
- System/network administrator.

AUBURN

Requirement Inconsistency

- Most **large software systems** address wicked problems
- Problems which are so **complex** that they can **never be fully understood** and
  where understanding **evolves** during the system development
- Therefore, requirements are normally both **incomplete**
  and **inconsistent**

AUBURN

Reasons for Requirement Inconsistency

- Large software systems must improve the current situation.
  It is **hard to anticipate the effects**
  that the new system will have on the organization
- **Different users** have different requirements and priorities.
  There is a constantly
  shifting **compromise** in the requirements
- System **end-users** and organizations who **pay** for
  the system have different requirements
- Prototyping is often required to clarify requirements

AUBURN

COMP 5710/6710: Software Quality Assurance

Akond Rahman, PhD
Department of Computer Science and Software Engineering (CSSE)
akond@auburn.edu
https://akondrahman.github.io

AUBURN

What is generative AI?

- The class of AI models that emulate the structure and characteristics of input data in order to
  generate derived synthetic content. This can include images,
  videos, audio, text, and other digital content.

AUBURN

Why Generative AI for Quality Assurance?

https://onlinescientificresearch.com/articles/generative-ai-for-vulnerability-management-a-blueprint.pdf
https://www.tenable.com/blog/how-to-discover-analyze-and-respond-to-threats-faster-with-generative-ai
https://swimlane.com/blog/how-can-generative-ai-be-used-in-cybersecurity/
https://developer.nvidia.com/blog/applying-generative-ai-for-cve-analysis-at-an-enterprise-scale/
https://arxiv.org/abs/2506.20488

AUBURN

Why Generative AI for Quality Assurance?

https://www.darkreading.com/vulnerabilities-threats/generative-ai-exacerbates-software-supply-chain-risks
https://www.zscaler.com/cxorevolutionaries/insights/ai-software-supply-chain-risks-prompt-new-corporate-diligence
https://versa-networks.com/blog/the-rise-of-slopsquatting-a-new-software-supply-chain-threat/
https://www.extrahop.com/blog/amid-rising-genaI-hacking-hysteria-supply-chain-most-at-risk
https://www.paloaltonetworks.com/cyberpedia/generative-ai-security-risks

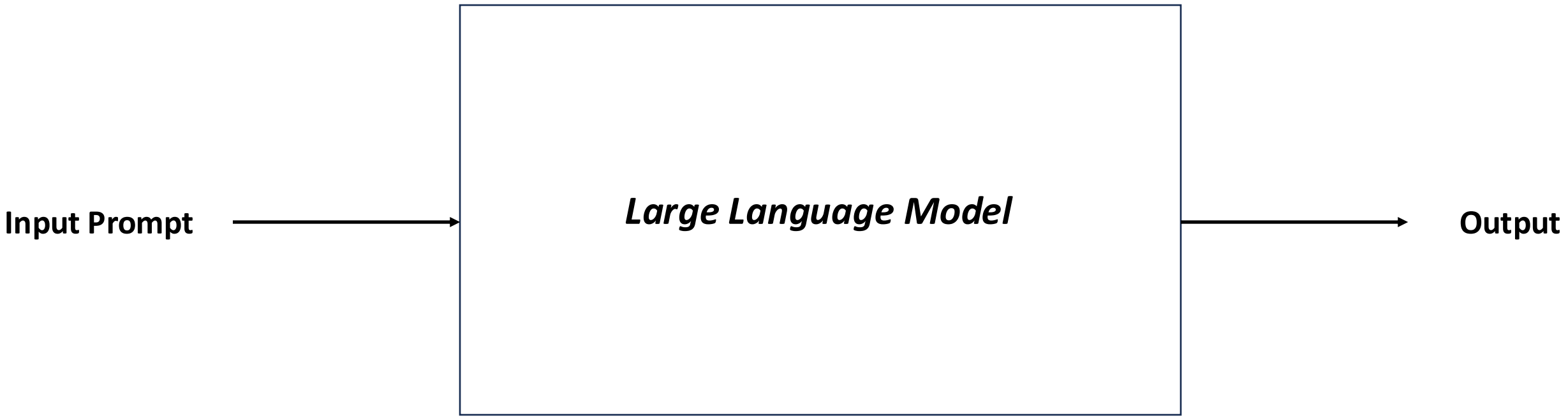
AUBURN

Why Generative AI for Me?

https://github.blog/ai-and-ml/generative-ai/what-developers-need-to-know-about-generative-ai/
https://www.sei.cmu.edu/blog/perspectives-on-generative-ai-in-software-engineering-and-acquisition/
https://www.wired.com/story/meta-ai-job-interview-coding/

AUBURN

Large Language Models (LLMs)



AUBURN

Input Prompts in LLMs

- Zero shot
- Few shot
- Chain of thought

AUBURN

Input Prompts in LLMs

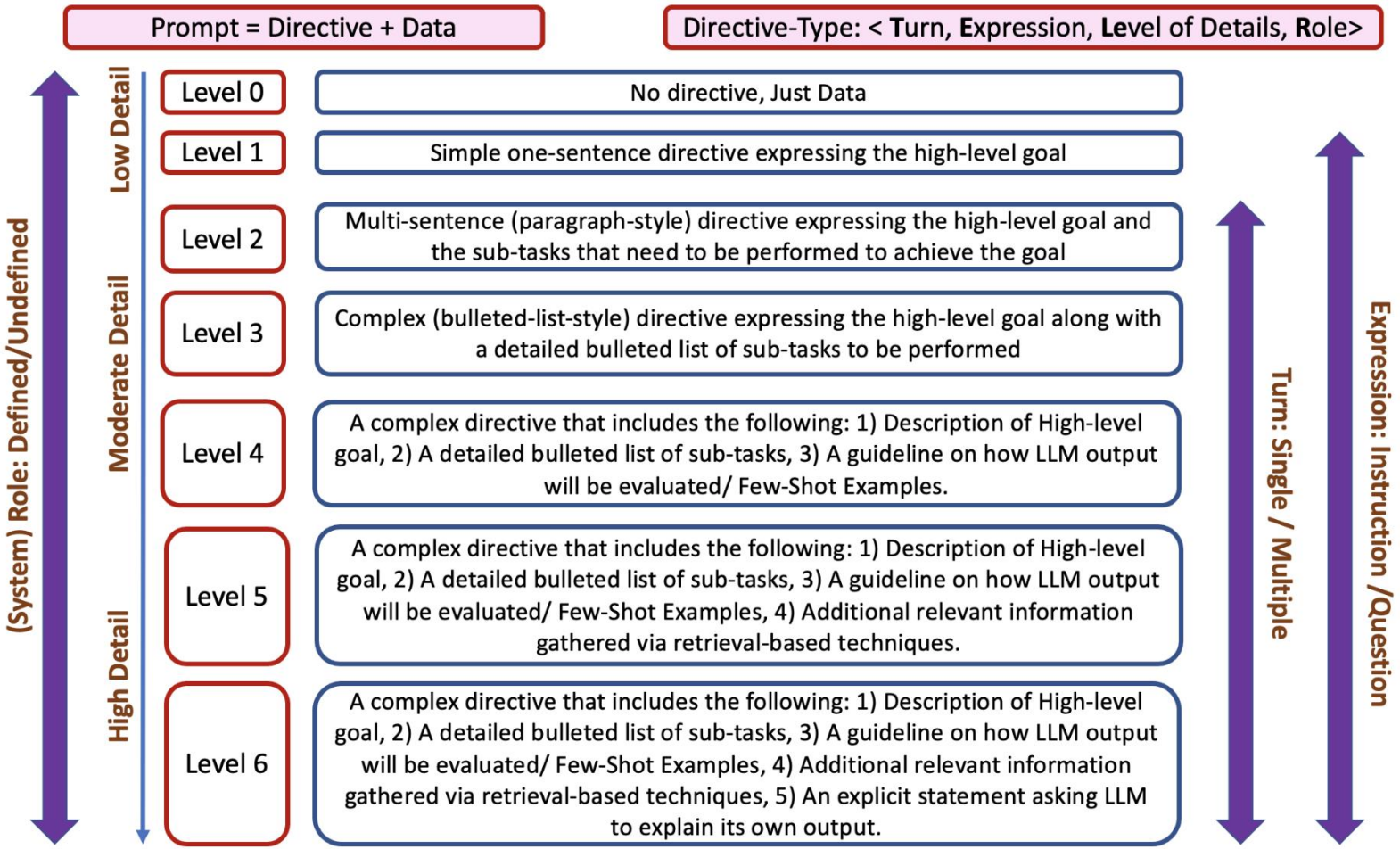


Figure 1: Proposed Prompt Taxonomy: **TELeR** (<**T**urn, **E**xpression, **Le**vel of Details, **R**ole>)

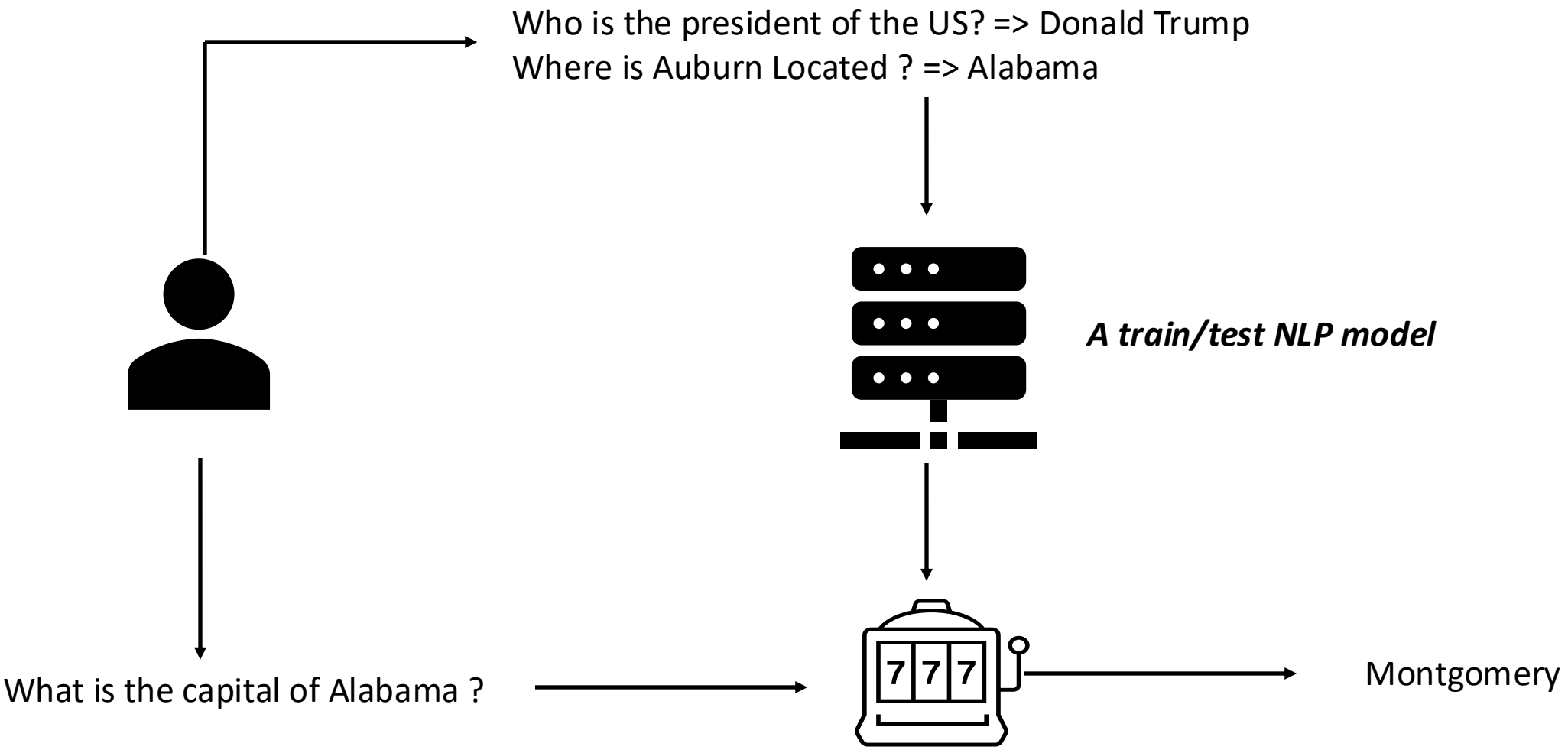
AUBURN

Generative AI Tools

- https://gemini.google.com/
- https://claude.ai/
- https://chatgpt.com/

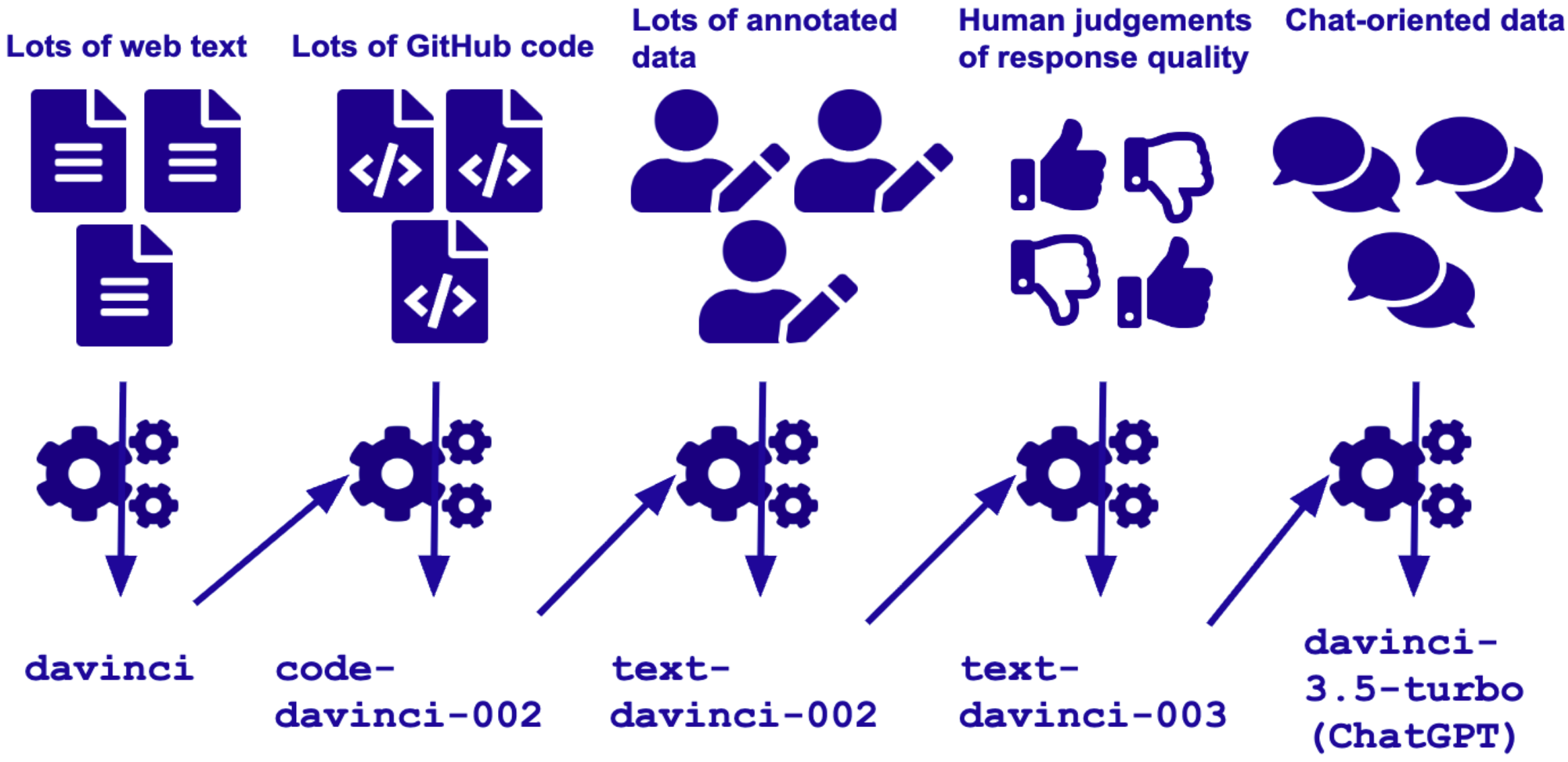
AUBURN

Evolution to LLMs

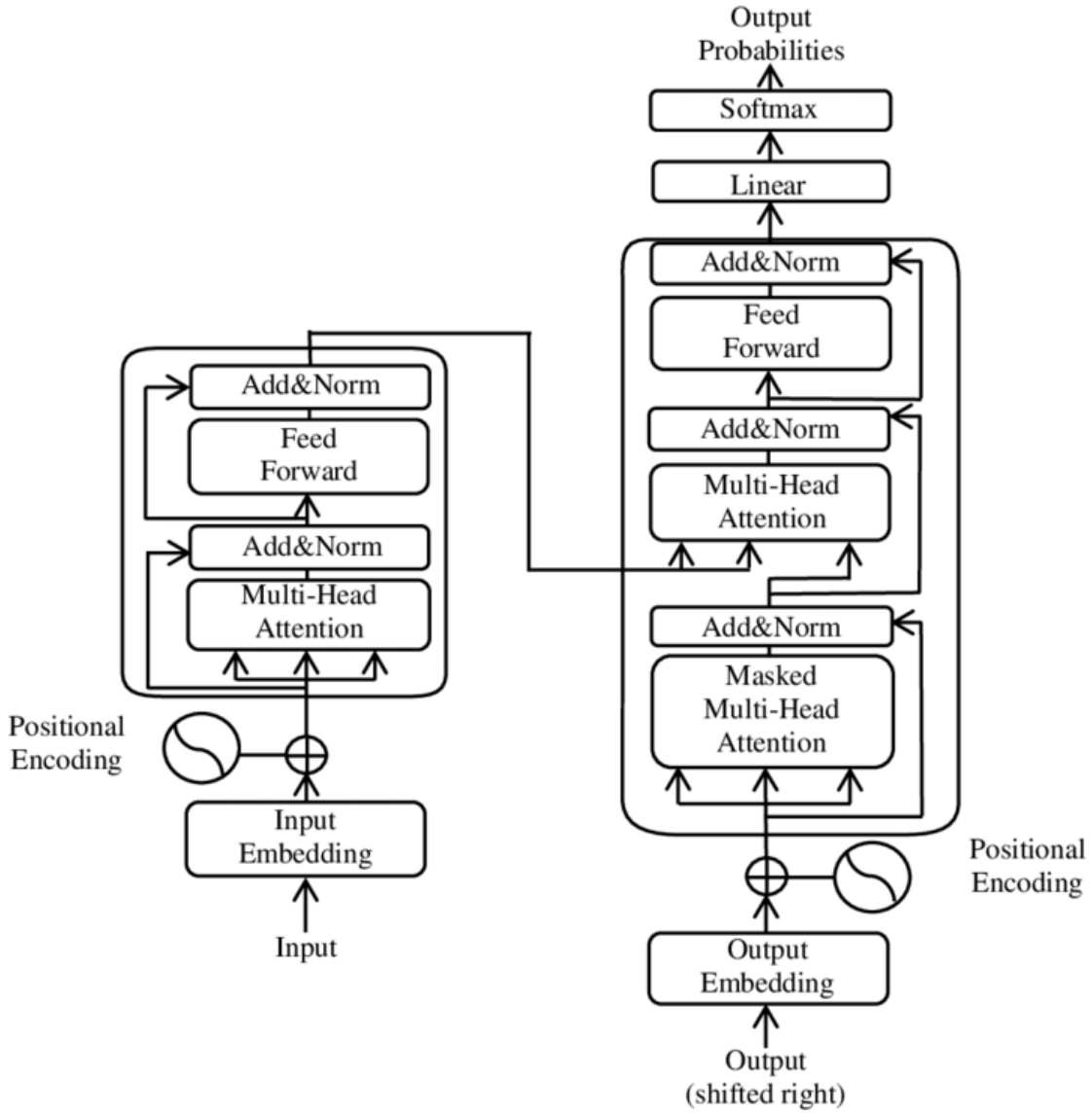


AUBURN

Evolution to LLMs



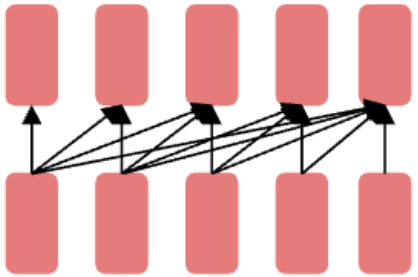
Inside an LLM



https://arxiv.org/pdf/1706.03762

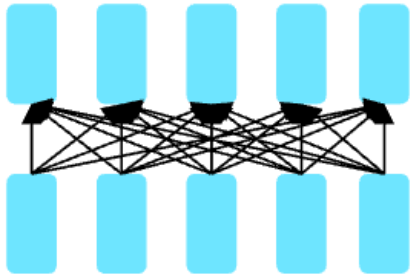
AUBURN

Inside an LLM (Contd.)



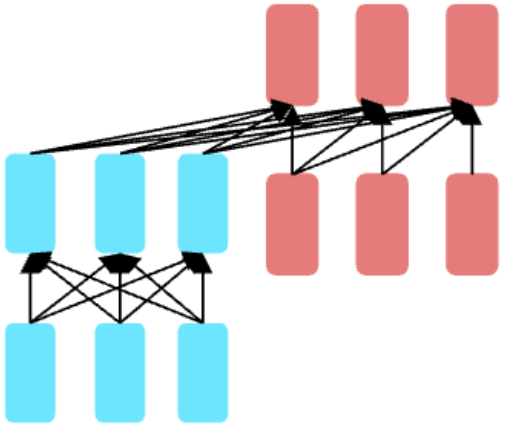
Decoders

GPT, Claude, Llama Mixtral



Encoders

BERT family, HuBERT



Encoder-decoders

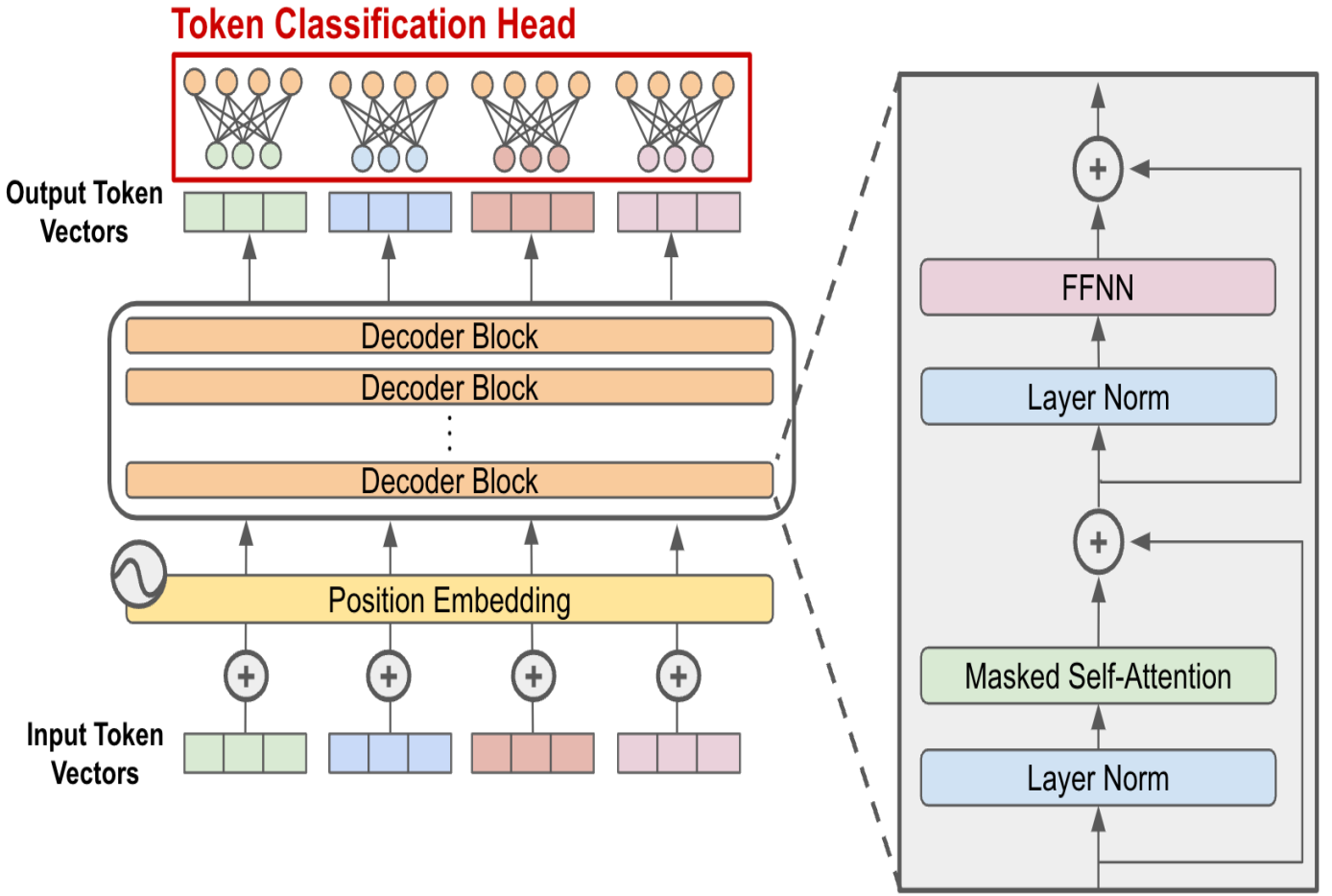
Flan-T5, Whisper

https://web.stanford.edu/~jurafsky/slp3/slides/LLM24aug.pdf

AUBURN

Inside an LLM (Contd.)

A decoder LLM
generates output text one token at a time, building a coherent sequence
based on the input and previously generated tokens

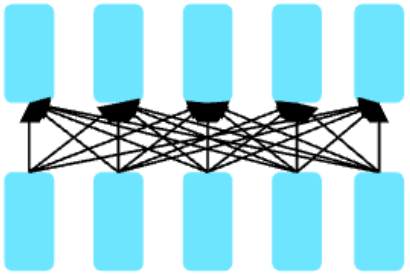


https://web.stanford.edu/~jurafsky/slp3/slides/LLM24aug.pdf

AUBURN

Inside an LLM (Contd.)

Encoders



Many varieties!

- Popular: Masked Language Models (MLMs)
- BERT family

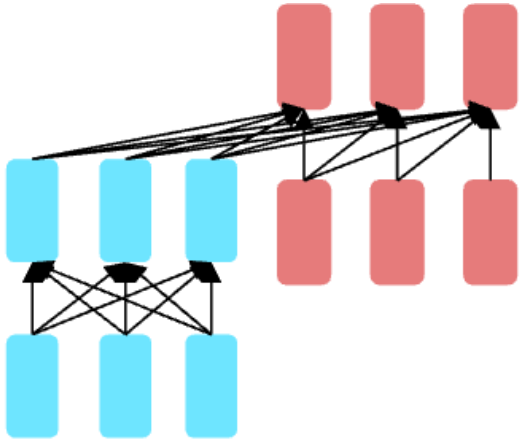
- Trained by predicting words from surrounding words on both sides
- Are usually **finetuned** (trained on supervised data) for classification tasks.

AUBURN

https://web.stanford.edu/~jurafsky/slp3/slides/LLM24aug.pdf

Inside an LLM (Contd.)

Encoder-Decoders



- Trained to map from one sequence to another
- Very popular for:
  - machine translation (map from one language to another)
  - speech recognition (map from acoustics to words)

https://web.stanford.edu/~jurafsky/slp3/slides/LLM24aug.pdf

COMP 5710/6710: Software Quality Assurance

Akond Rahman, PhD
Department of Computer Science and Software Engineering (CSSE)
Shelby Center 3104
akond@auburn.edu
https://akondrahman.github.io

AUBURN

Configuration Management Practices

Traditional approach

Infrastructure as code

AUBURN

Traditional Approach

AUBURN

Software Configuration Management (SCM)

"Configuration management … is the discipline of identifying the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system life cycle."

AUBURN

Software Configuration Management (SCM)

New versions of software systems are created as they change:

- For different machines/OS;
- Offering different functionality;
- Tailored for particular user requirements.

Configuration management is concerned with managing evolving software systems:

- System change is a team activity;
- CM aims to control the costs and effort involved in making changes to a system.

AUBURN

Software Configuration Management (SCM)

Involves the development and application of procedures and standards to manage an evolving software product.
CM may be seen as part of a more general quality management process.
When released to CM, software systems are sometimes called *baselines* as they are a starting point for further development.

AUBURN

Software Configuration Management (SCM)

Version control:

Commonly used in engineering and software development.

Changes identified by incrementing an associated number or
letter code - “revision number”, or “revision level”.

Revision numbers (levels) are associated historically with the person making the change.

AUBURN

SCM: Check In

Copies a file into the database.

The version you see is (usually) the latest version that was checked in.

The old version is kept and can be accessed later.

AUBURN

SCM: Check Out

Copies the database file to your personal folder and raises a flag on the database copy.

Need to check out before checking in.

AUBURN

Misconfigurations

- Misconfigurations account for 82% of security vulnerabilities.
- Exposure of 750,000 birth certificate applications in the US on an AWS storage bucket in 2019
- Misconfigured cloud storage bucket exposing hundreds of thousands of mobile phone bills for AT&T, Verizon and T-Mobile subscribers due to human error and misconfiguration.

AUBURN

Misconfigurations

- https://threatpost.com/payment-api-exposes-payment-data/174825/
- https://threatpost.com/misconfiguration-university-wifi-login-credentials/175157/
- https://www.toolbox.com/it-security/cyber-risk-management/news/aws-vulnerable-to-ransomware-attacks/

AUBURN

Example Misconfigurations

- default configurations
- open cloud storage
- insecure HTTP requests
- verbose error messages

AUBURN

Example Misconfigurations

- If Directory listing is not disabled on the server and if attacker discovers the same then the attacker can simply list directories to find any file and execute it.
- App servers usually come with sample apps that are not well secured. If not removed from production server would result in compromising your server.

AUBURN

How to Study and Assess Misconfigurations

Threat Agents	• Anonymous external attackers as well as users with their own accounts that may attempt to compromise the system.
Attacker's Approach	• Accesses default accounts, unused pages, unpatched flaws, unprotected files and directories to gain unauthorized access
Security Weakness	• Can happen at any level - platform, web server, application server, database, framework, and custom code.
How to Spot	• Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.
Technical Impact	• All of your data could be stolen or modified slowly over time. • Recovery Costs - Expensive
Business Impact	• The system could be completely compromised without the knowledge of the Application owners.

AUBURN

How to Mitigate Misconfigurations

- Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform.
- Secure settings should be defined, implemented, and maintained, as defaults are often insecure.

AUBURN

How to Mitigate Misconfigurations

- Build a strong application architecture that provides secure and effective separation of components.
- Run audits and scans frequently and periodically to help identify potential security misconfigurations or missing patches.
- Maintain a well-structured and maintained development cycle. This will help ensure the security testing of the application during the development phase.

AUBURN

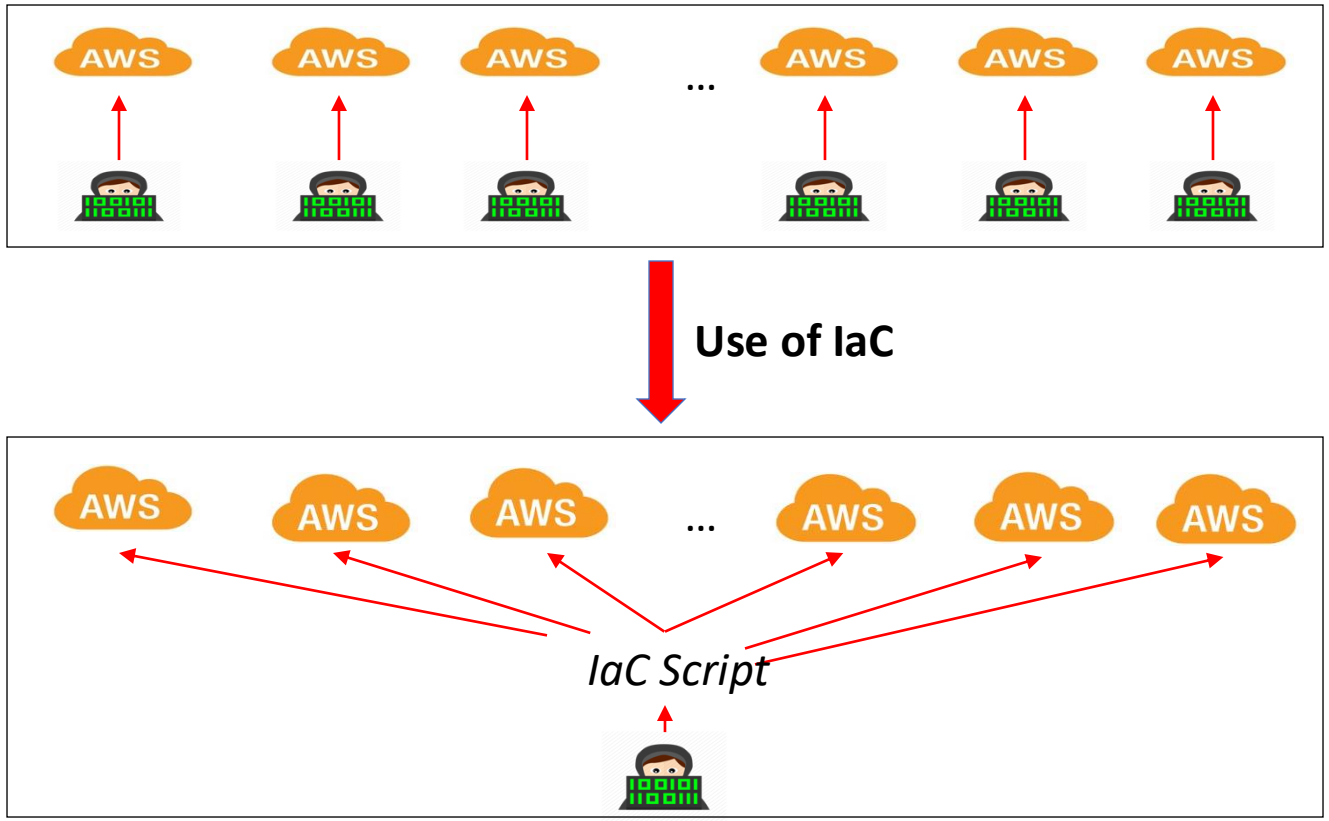
Configuration Management Practices

Modern Approach

Puppet script

AUBURN

Automated Configuration Management



AUBURN

Automated Configuration Management

Version control

Automation script

Quality assurance

AUBURN

Automated Configuration Management: Languages

Ansible

Chef

Puppet

Kubernetes

AUBURN

Automated Configuration Management: Languages

```
include ssh::ldap

file { "/var/tmp/log":
        ensure => "present",
        owner => "root",
        group => "root",
        mode => '0644',
}


exec { "ceph-injectkey-${name}":
        command => "/bin/true -set -ex -ceph auth add ${name}
            ↪ --in-file=${keyring_path}",
        require => [ Package['ceph'], Exec["ceph-key-${name}"], ],
        logoutput => true,
        unless => "ceph auth get ${name} | grep ${secret}",
}
```

Include

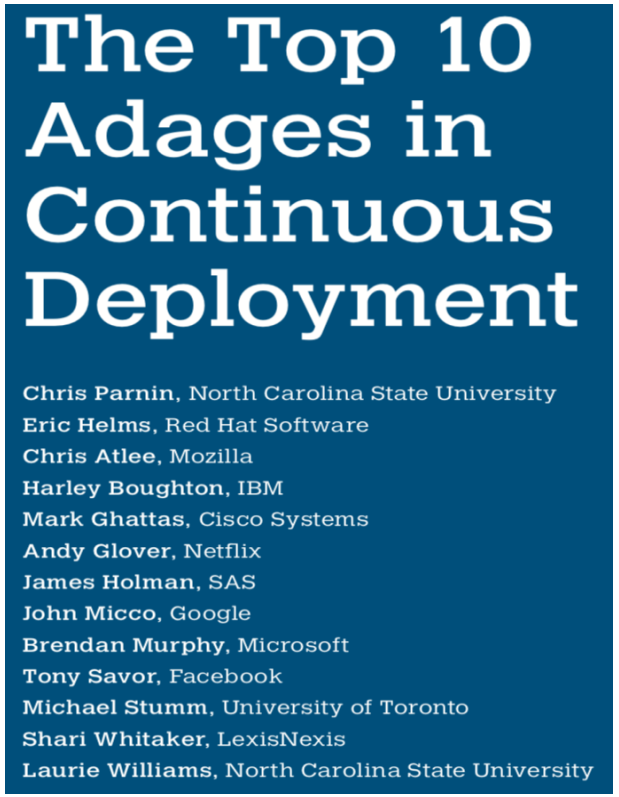
File

File mode

Command

AUBURN

Automated Configuration Management: New Normal



"The new normal is that organizations should treat managing configuration the same as managing features and code"

AUBURN

Automated Configuration Management: State-based Approach

1. Get the state of system
2. Get the state represented in the configuration file
3. Compare
4. Make the change if there is a difference

AUBURN

Automated Configuration Management: Use Cases

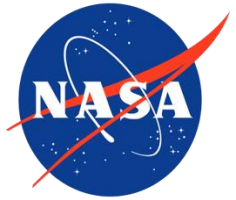
Mozilla

GitHub

Openstack

Intercontinental Exchange

Ambit Energy



NASA

AUBURN

Automated Configuration Management: Benefits

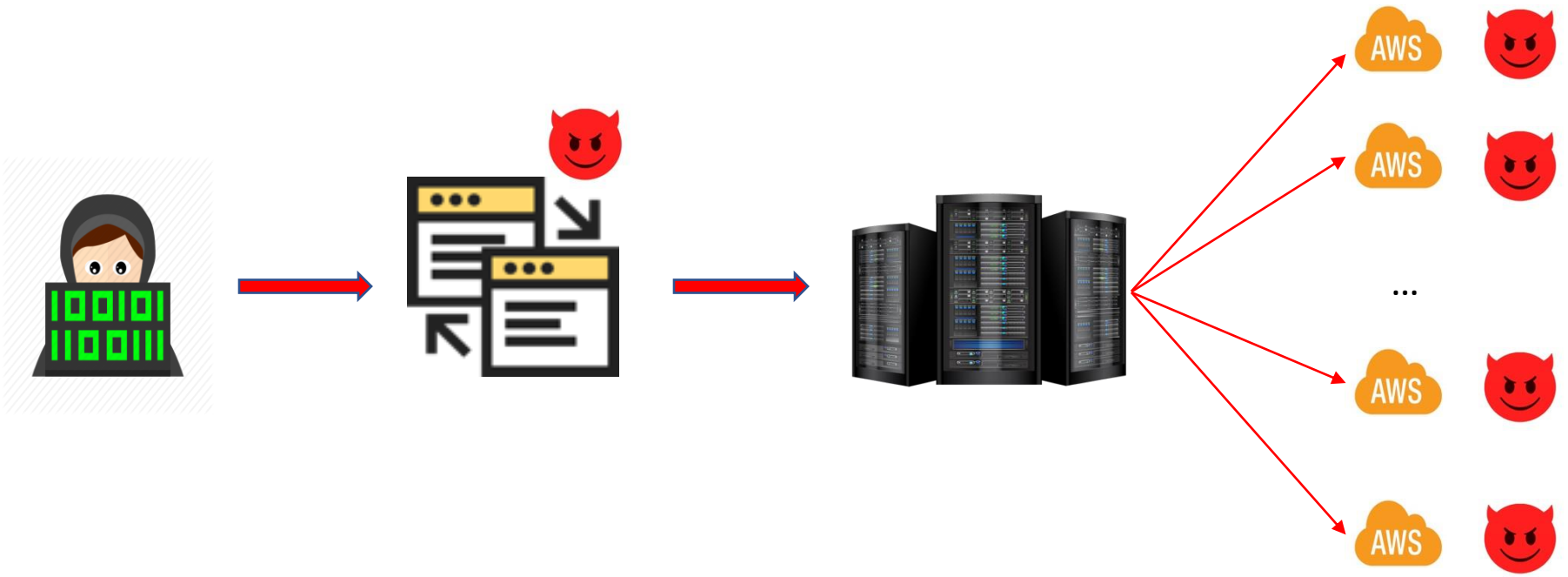
Reduced provisioning time
1-2 days to 21 minutes

Deployment frequency
1200x improvement

Reduced patch release time
multiple days to 1 hour

AUBURN

Automated Configuration Management: Quality Assurance



Without quality assurance

AUBURN

Automated Configuration Management: Quality Assurance

GitHub's DNS Outage

AUBURN

Automated Configuration Management: Quality Assurance

Incident documentation/20170118-Labs

< Incident documentation

Contents [hide]

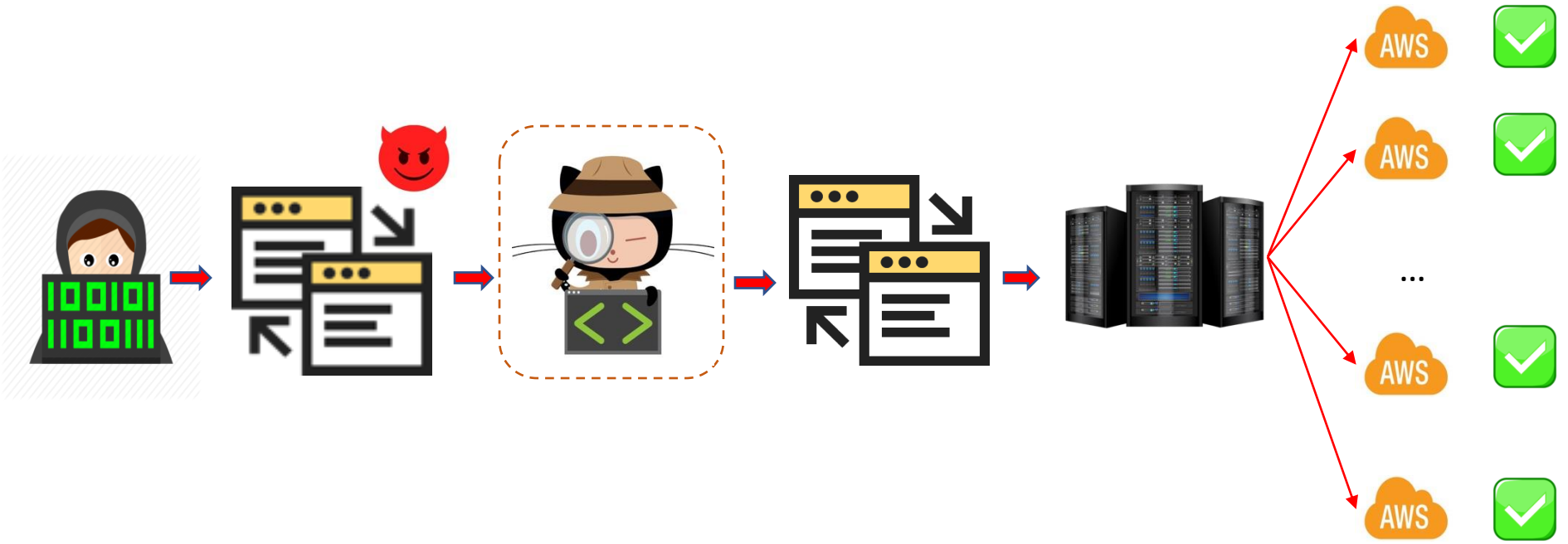
Summary

~270 user home directories across labs instances that didn't have NFS mounted (non-tools/maps instances) got erased as an unintended side effect of the planned NFS migration(Task T154336) for moving rest of labs projects on NFS to the secondary NFS cluster. This happened due to merging an erroneous puppet patch that made /home a symlink to the mount path across all instances, instead of just on nodes where /home was on NFS. We were able to restore most of the data using puppet filebucket restore mechanisms, but a few home directories, and some executable file permissions have been permanently lost.

Wikimedia Commons

AUBURN

Automated Configuration Management: Quality Assurance



With quality assurance

AUBURN

Automated Configuration Management: Common Defects

Conditional
Configuration data
Dependency
Documentation
Idempotency
Security
Service
Syntax

AUBURN

Automated Configuration Management: Impacted Infras.

- CI
- Communication Platform
- Containerization
- Data Storage
- File
- Networking

AUBURN

COMP 5710/6710: Software Quality Assurance

Akond Rahman, PhD
Department of Computer Science and Software Engineering (CSSE)
Shelby Center 3104
akond@auburn.edu
https://akondrahman.github.io

AUBURN

Static Analysis

Abstraction

- Elide details of a specific implementation.
- Capture semantically relevant details; ignore the rest.

Programs as data

- Programs are just trees/graphs!

Static vs Dynamic Analysis

Static analysis

- Analyze code without executing it.
- Systematically follow an abstraction.
- Find bugs/enforce specifications/prove correctness.
- Often correctness/security related.

Dynamic analysis

- Execute code
- Instrument program
- Abstract parts of the trace.
- Find bugs/enforce stronger specifications/debugging/profiling.
- Often memory/performance/concurrency/security related.

Static Analysis Tools: Usefulness

Defects that result from inconsistently following simple design rules.

Security: Improperly validated input.

Memory safety: Null dereference, uninitialized data.

API Protocols: Device drivers; real time libraries; GUI frameworks.

Exceptions: Arithmetic/library/user-defined

Encapsulation: Accessing internal data, calling private functions.

Key: check compliance to simple, mechanical design rules

AUBURN

Static Analysis Tools: Limitations

- Array Bounds, Interrupts
  - Testing
    - Errors typically on uncommon paths or uncommon input
    - Difficult to exercise these paths
  - Inspection
    - Non-local and thus easy to miss
      - Array allocation vs. index expression
      - Disable interrupts vs. return statement
- Finding Race Conditions
  - Testing
    - Cannot force all interleaving
  - Inspection
    - Too many interleavings to consider
    - Check rules like “lock protects x” instead
      - But checking is non-local and thus easy to miss a case

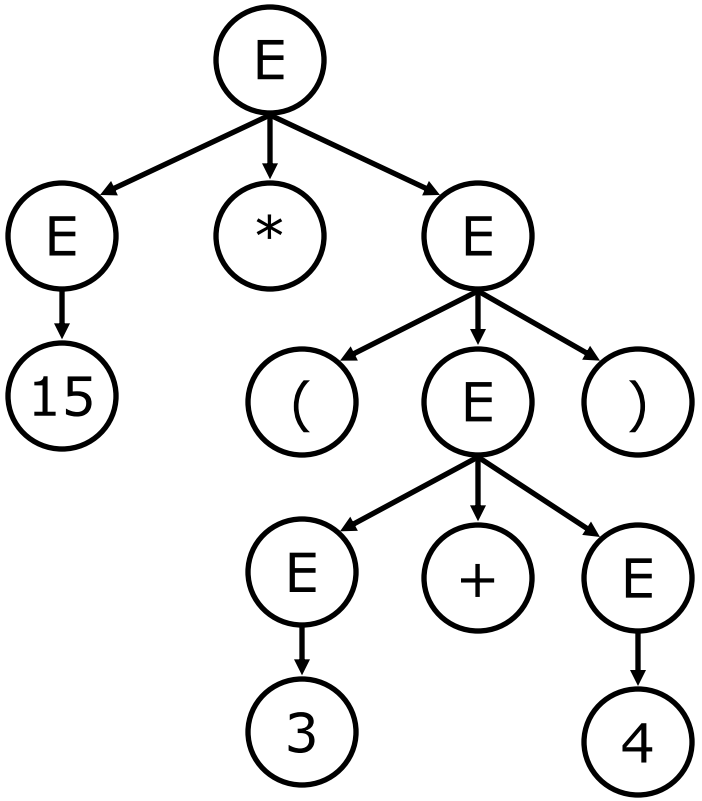
Static Analysis Tools: Sophistication

Pattern identification (FindBugs, Lint)
Type checking
Dataflow analysis
Model checking
Formal reasoning

Sophistication

AUBURN

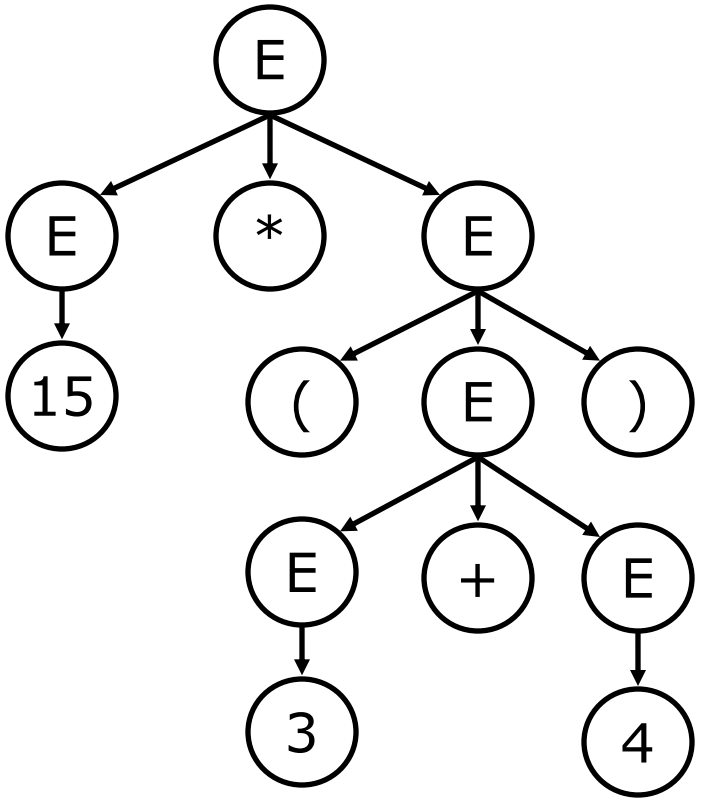
Static Analysis Tools: AST



- Parse trees encodes the grammatical structure of the source program
- However, they contain a lot of unnecessary information
- What is essential here?

AUBURN

Static Analysis Tools: AST



For the compiler to understand an expression, it only need to know operators and operands

punctuations, parentheses, etc. are not needed

Similar for statements, functions, etc.

AUBURN

Static Analysis Tools: AST vs Parse Trees

Concrete Syntax i.e. parse tree is needed for parsing

includes punctuation symbols, depends on the format of the input

Abstract Syntax is simpler, more convenient internal representation

clean interface between the parser and

the later phases of the compiler

AUBURN

Static Analysis Tools: Data Flow Analysis

Track a bug/vulnerability as data across a program

Taint Tracking: Available Expression

An expression e is available at program point p if

e is computed on every path to p, and

the value of e has not changed since the last time e is computed on p

Optimization

If an expression is available, need not be recomputed

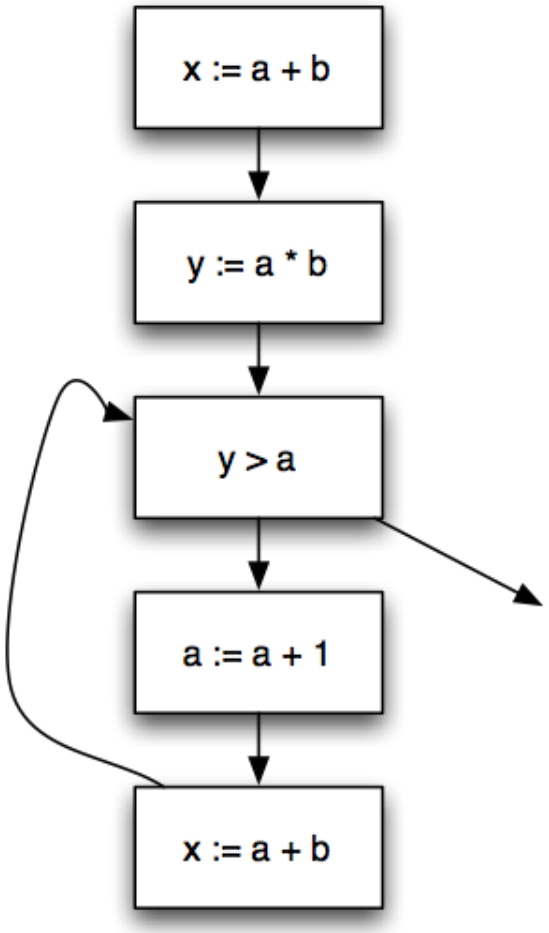
(At least, if it's still in a register somewhere)

Taint Tracking: Available Expression

Is expression e available?

Example Facts:

- a + b is available
- a * b is available
- a + 1 is available



AUBURN

Taint Tracking: Available Expression Example

1. `a = y+2`	`y+2` is available after statement 1
2. `z = x+w`	`y+2,x+w`
3. `x = y+2`	`y+2` (`y+2` is available in `a,` but `x+w` is not)
4. `z = b+c`	`y+2,b+c`
5. `b = y+2`	(`y+2` is available in `a,` but `b+c` is not)

AUBURN

Taint Tracking: Def-use Chains

A def-use association is a triple (x, d, u,), where:

x is a variable,
d is a node containing a definition of x,
u is either a statement or predicate node containing a use of x,

and there is a sub-path in the flow graph from d to u with no other definition of x between d and u.

AUBURN

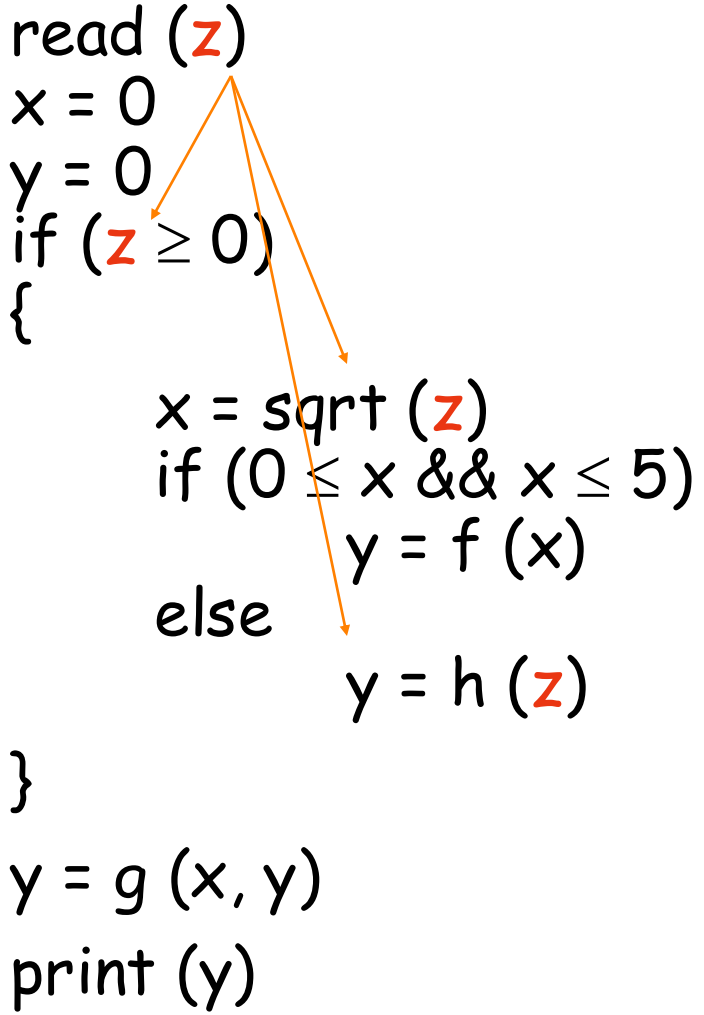
Taint Tracking: Example of Def-use Chains

What are all the *def-use associations* for the program below?

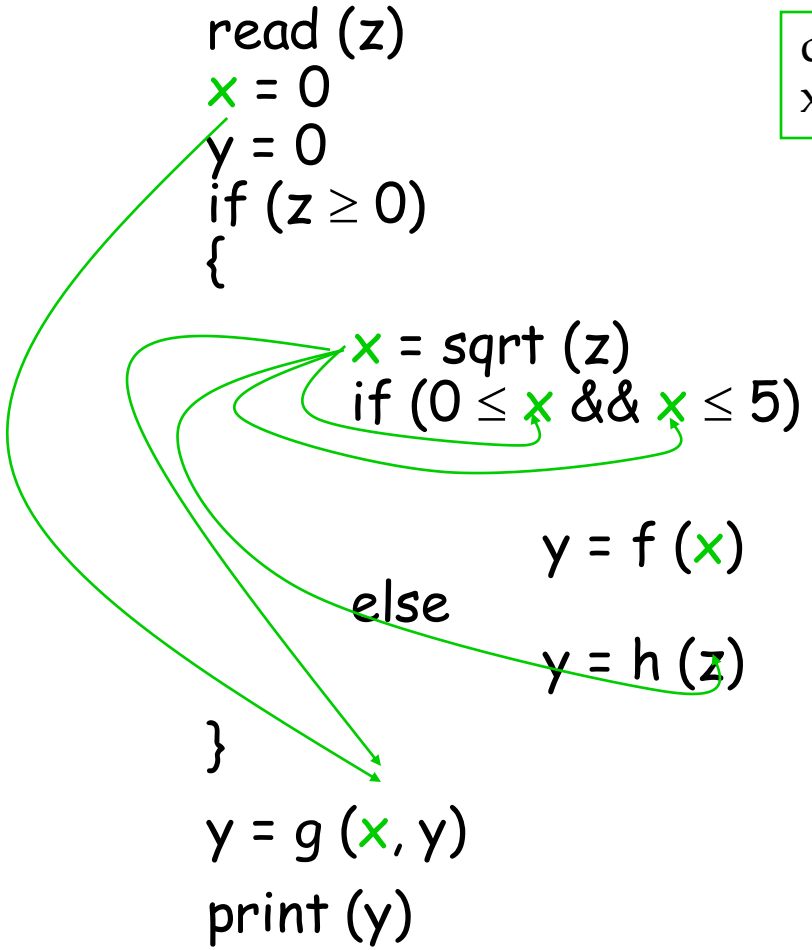
```
read (z)
x = 0
y = 0
if (z ≥ 0)
{
        x = sqrt (z)
        if (0 ≤ x && x ≤ 5)
                y = f (x)
        else
                y = h (z)
}
y = g (x, y)
print (y)
```

Taint Tracking: Example of Def-use Chains

def-use associations for variable z.



Taint Tracking: Example of Def-use Chains



def-use associations for variable x.

Taint Tracking: Example of Def-use Chains

```
read (z)
x = 0
y = 0
if (z ≥ 0)
{
        x = sqrt (z)
        if (0 ≤ x && x ≤ 5)
                y = f (x)
        else
                y = h (z)
}
y=g (x, y)
print (y)
```

def-use associations for variable y.

Taint Tracking: Example

```
 private static class Internal {
   private String s;
   public Internal(String s) {
      this.s = s;
   }
   public String printString() {
     return s;
   }
}
Internal i1 = new Internal(s1); // s1 is tainted
writer.println(i1.printString())
```

Taint Tracking: Example

```
private static class Internal {
    private String s;
    public Internal(String s) {
        this.s = s;
    }
    public String getString() {
        return s;
    }
}
Internal i1 = new Internal(s1); // s1 is tainted
writer.println(i1.getString())
```

Taint Tracking: Example

```
protected void doGet(HttpServletRequest req,
HttpServletResponse resp) throws IOException {
 try {
 ...
 } catch (Exception e) {
   resp.getWriter().println(e);
 }
}
```

COMP 5710/6710: Software Quality Assurance

Akond Rahman, PhD
Department of Computer Science and Software Engineering (CSSE)
akond@auburn.edu
https://akondrahman.github.io

AUBURN

Integration

- Combining 2 or more software units
  - often a subset of the overall project (!= system testing)

- Why do software engineers care about integration?

  - new problems will inevitably surface
    - many systems now together that have never been before

  - if done poorly, all problems present themselves at once
    - hard to diagnose, debug, fix

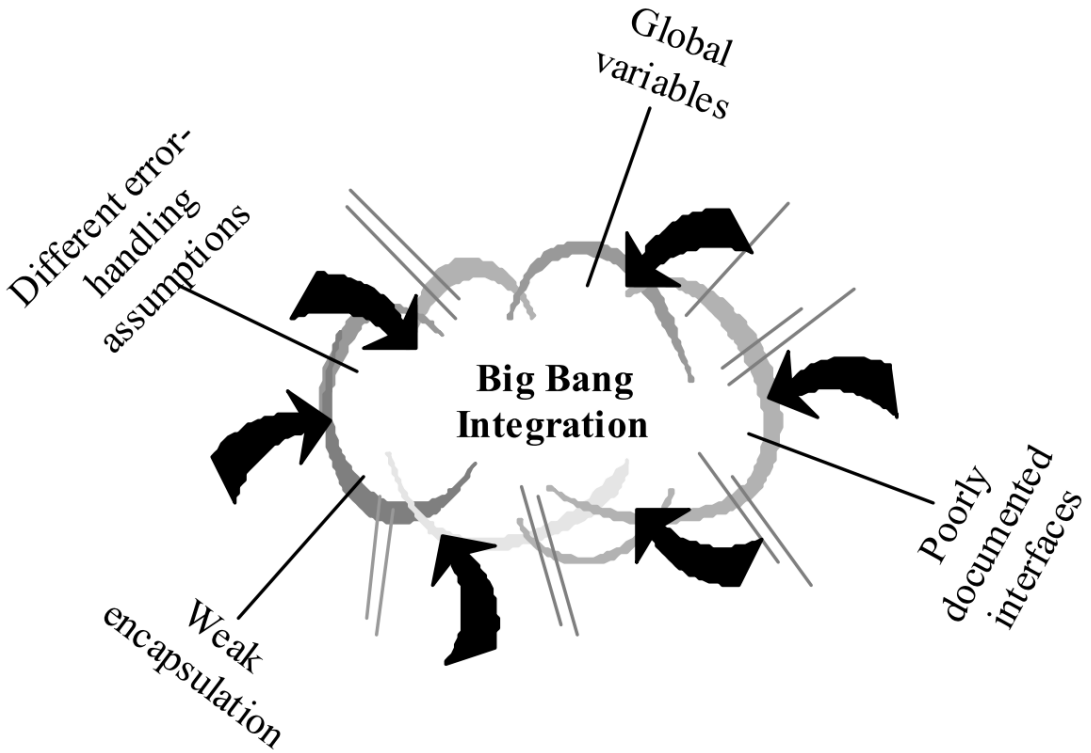
  - cascade of interdependencies
    - cannot find and solve problems one-at-a-time

AUBURN

Phased Integration

phased ("big-bang") integration:

- design, code, test, debug each class/unit/subsystem separately
- combine them all
- pray



AUBURN

Incremental Integration

- **incremental integration**:
  - develop a functional "skeleton" system (i.e. ZFR)
  - design, code, test, debug a small new piece
  - integrate this piece with the skeleton
    - test/debug it before adding any other pieces

AUBURN

Benefits of Incremental Integration

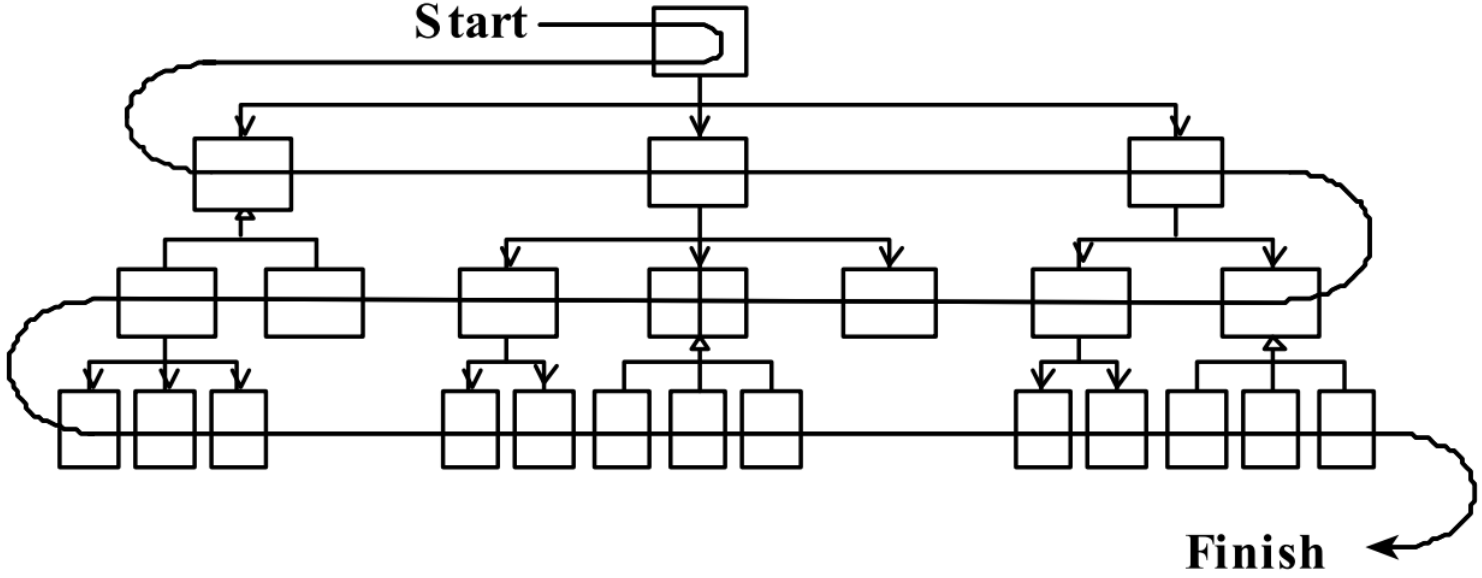
- Benefits:
  - Errors easier to isolate, find, fix
    - reduces developer bug-fixing load
  - System is always in a (relatively) working state
    - good for customer relations, developer morale
- Drawbacks:
  - May need to create "stub" versions of some features that have not yet been integrated

AUBURN

Top Down Integration

Start with outer UI layers and work inward

- must write (lots of) stub lower layers for UI to interact with
- allows postponing tough design/debugging decisions (bad?)

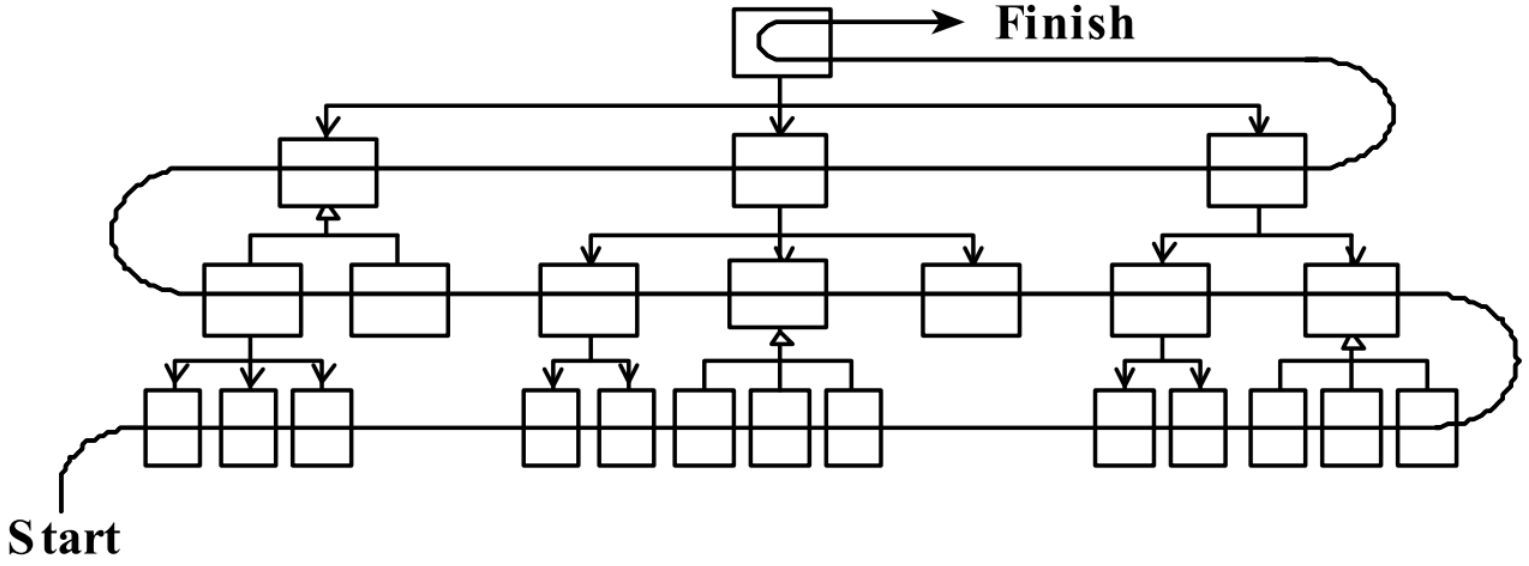


AUBURN

Bottom Up Integration

Start with low-level data/logic layers and work outward

- must write test drivers to run these layers
- won't discover high-level / UI design flaws until late



AUBURN

Continuous Integration

- Pioneered by Martin Fowler; part of Extreme Programming
- Principles:
  - maintain a single source repository
  - automate the build
  - everyone commits to mainline every day
  - every commit should build mainline on an integration machine
  - keep the build fast
  - test in a clone of the production environment
  - make it easy for anyone to get the latest executable
  - everyone can see what's happening

Continuous Integration: Daily Build

"Automate the build."

daily build: Compile working executable on a daily basis

- allows you to test the quality of your integration so far
- helps morale; product "works every day"; visible progress
- best done *automated* or through an easy script
- quickly catches/exposes any bug that breaks the build

Continuous Integration (CI) server: An external machine that automatically pulls down your latest repo code and fully builds all resources.

- If anything fails, contacts your team (e.g. by email).
- Ensures that the build is never broken for long.

AUBURN

Continuous Integration: Automated Tests

"Make your build self-testing."

automated tests: e.g. Tests that can be run from the command line on your project code at any time.

- can be unit tests

smoke test: A quick set of tests run on the daily build.

- NOT exhaustive; just sees whether code "smokes" (breaks)
- used (along with compilation) to make sure daily build runs

Continuous Integration: Daily Commits

"Everyone commits to the mainline every day."

daily commit: Submit work to main repo at end of each day.

Idea: Reduce merge conflicts; avoid later integration issues.

This is the key to "continuous integration" of new code.

Caution: Don't check in faulty code (does not compile, does not pass tests) just to maintain the daily commit practice.

If your code is not ready to submit at end of day, either submit a coherent subset or be flexible about commit schedule.

Continuous Integration: Benefits

- Increased productivity
  - Enables shorter feedback cycle when changes are made
  - Code gets back into the hands of testers quickly
  - Frees the team to do more interesting and valuable work
  - Improves morale, making it easier to retain good developers
  - Enables more frequent releases with new features
- Improved quality
  - Makes it easier to find and remove defects because frequent integration and testing identifies bugs *as they are introduced.*
  - Multi-platform builds help in finding problems that may arise on some, but not all, versions of the supported platform.
- Reduced Risk
  - Reduces uncertainty greatly because at all times the team knows what works, what does not, and what the major issues are.

AUBURN

Open Source Continuous Integration

- Jenkins
- Travis CI
- GitHub
- GitLab
- Circle-CI

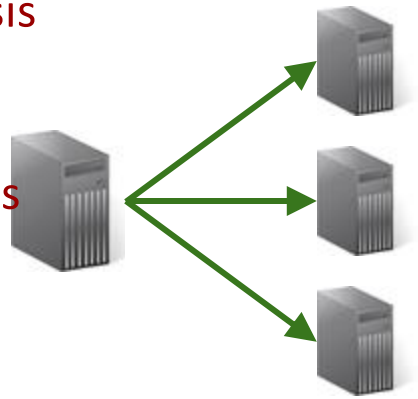
AUBURN

Concepts Related to Continuous Integration

Everything on 1 server
- Possible, not ideal unless for special projects

Distributed builds
- faster, safer, scalable, multiple platforms, more expensive

CI Server:
- Poll version control repo
- Perform actions on scheduled basis
- Send emails
- Host web accessible dashboard
- Store and display history of builds



CI Slaves:
- Pure workhorse
- Various architectures
- Reduces build time
- Communication through built in ssh-client (or similar)
- Receives action from server
- Carries out action

AUBURN

Concepts Related to Continuous Integration

Testing

Absolutely crucial for CI!

Several forms of testing:

Unit Tests

- First step of testing
- Can be run on all commits
- During build
- Or after build

Integration tests

- Does my new code/changes break the rest of the build?
- Probably after build

AUBURN

Concepts Related to Continuous Integration

Feedback

Feedback can be many things

- Website widget
- Email notifications
- IRC bot
- Extreme feedback
  - Lamps
  - Ambient orbs
  - Retaliation - Foam missiles (seriously)

AUBURN

Continuous Integration: A Modern Outlook

How Continuous Integration looks today?

Multiple specific configurations of tools to develop, build, deploy, and manage an application.

Continuous Integration: A Modern Outlook

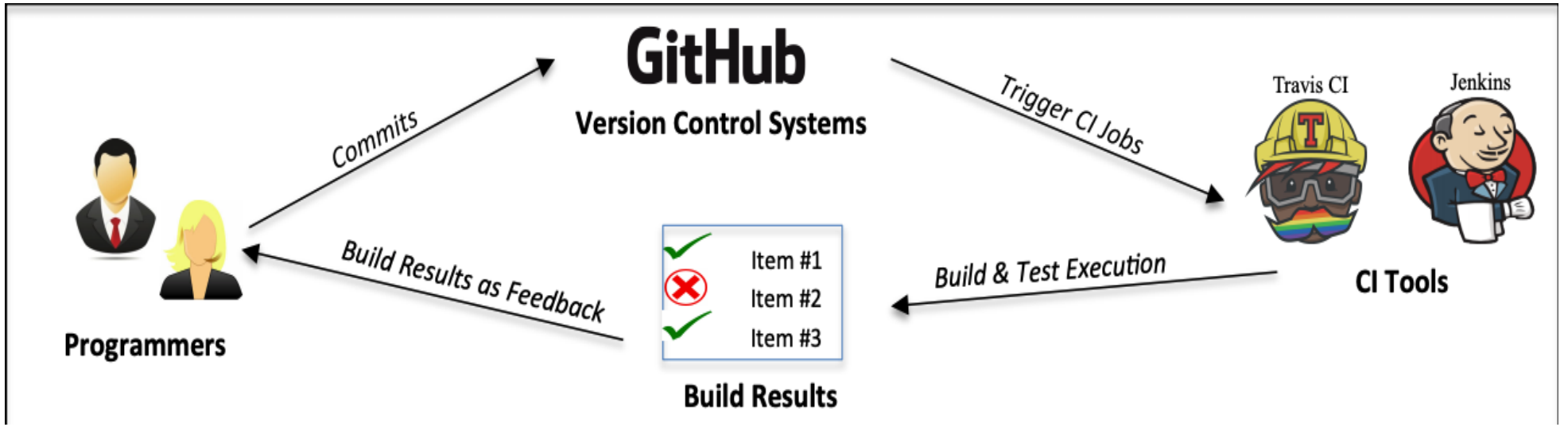
Pipeline:

- Automate builds, unit tests, deployments, and more.
- The stages in the pipeline can automatically build when you push changes to a linked repository and then deploy to one or more environments on the Cloud.
- You can also incorporate builds and deployments into your toolchains to connect builds to other tools.

AUBURN

Concepts Related to Continuous Integration

Workflow



AUBURN

Barriers to Continuous Integration

- Why doesn't every team already practice CI?
  - Troubleshooting failures
  - Automating build process
  - Setting up and maintaining a CI server
  - Details in Table 2 of https://dl.acm.org/doi/pdf/10.1145/3106237.3106270

AUBURN