RISK AND RESILIENCE BOOTCAMP

TEKsystems Global Services

WORKFORCE DEVELOPMENT

PROCESS TYPES

This section is an introduction into process models
This will include

- Agile and adaptive methodologies
- DevOps and DevSecOps
- CI/CD automation

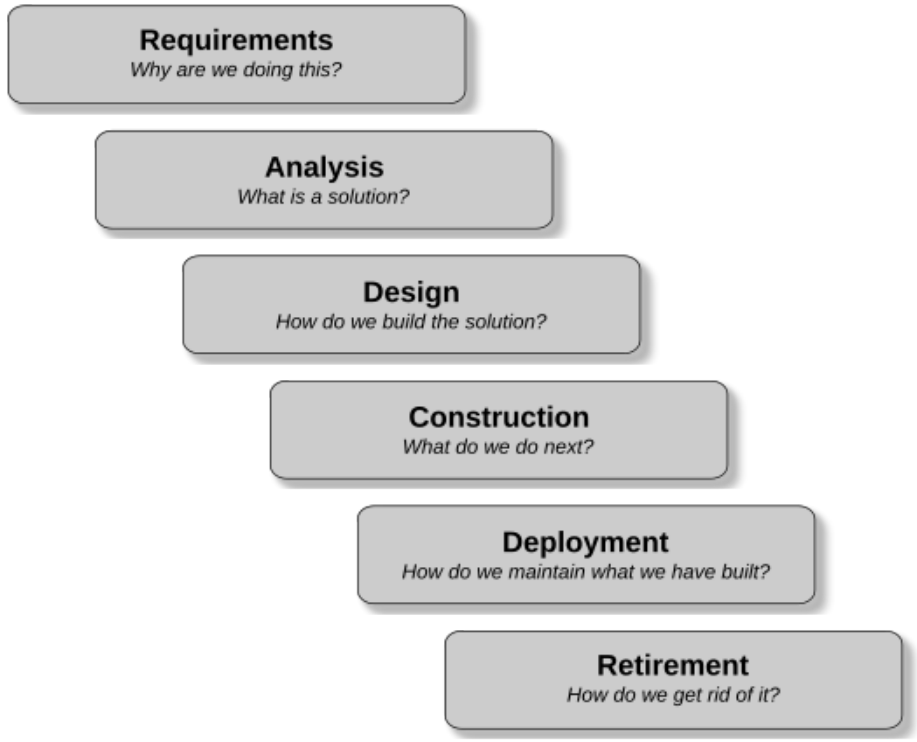
THE ENGINEERING CYCLE

The Engineering Cycle: a set of logical steps

- Each step builds on the previous one
- How we apply this cycle is a “process type”

Waterfall process types

- Completes each step in the process fully before moving on to the next step
- Also called a “predictive” process because given the full set of requirements and technical constraints, we can accurately predict the final product will be
- Common in engineering systems and high risk systems
  - Like nuclear reactor control software or airline navigation software
  - Or where requirements and technology don’t change over the lifetime of the project



IN REAL LIFE

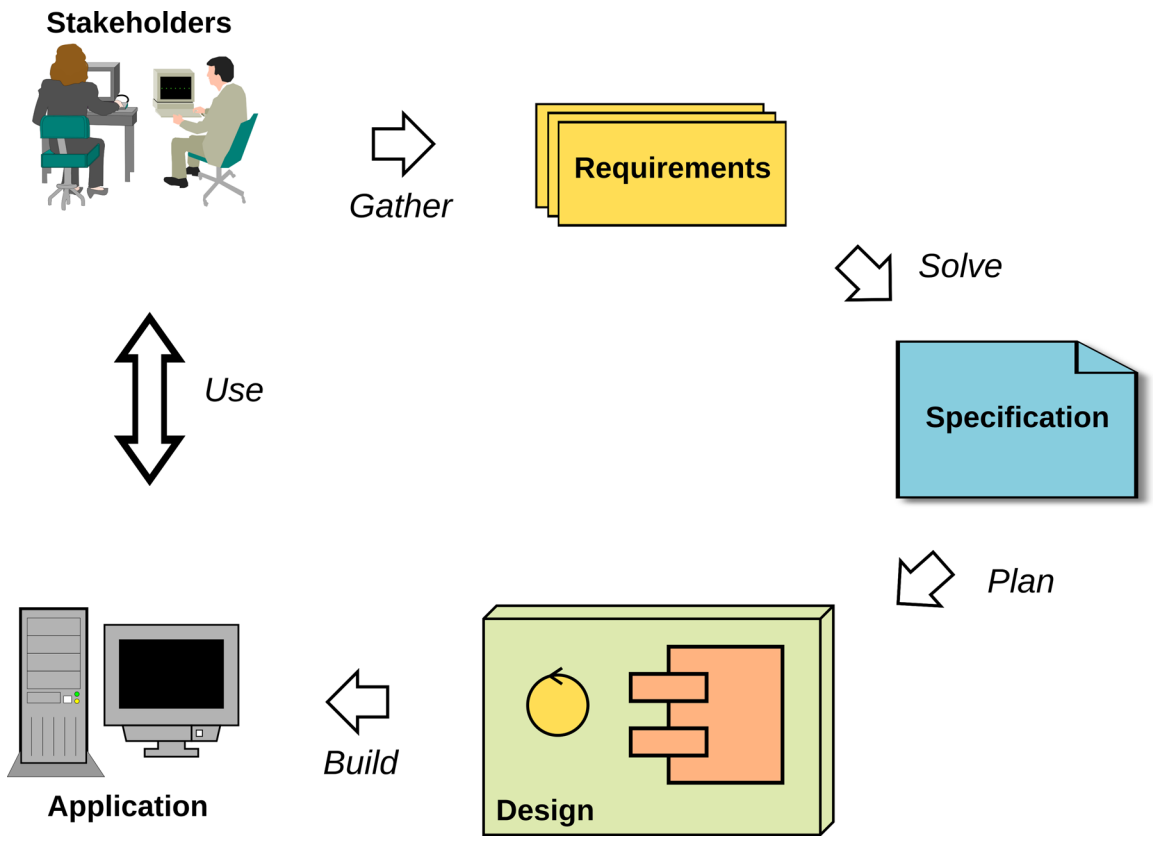
- For a lot of applications, the waterfall doesn't work
  - There is often too much uncertainty and variation at each stage of the engineering cycle
  - Results in a lot of re-work as we respond to unplanned for change
- These problems are not just software related
  - They were common across a variety of industries
  - Various alternatives to the waterfall approach were being experimented with
- Collectively, these are referred to as adaptive methodologies
  - The continuously adapt to uncertainty and variations
  - Essentially incorporating risk and resilience into a production process
  - Most notable of these is Scrum

RISK IN DEVELOPMENT

- Project risk
  - Risk of the project not being delivered successfully
  - Risk of building software that creates more operational risk or has poor resiliency
  - Risk of wasting resources by building the wrong thing or building it poorly
- Product risk
  - Risks that are created by using the project in production
  - Security vulnerabilities, problematic integration with legacy systems
- Business risk
  - A faulty product may disrupt business operations in unexpected ways
  - Risk analysis cannot be relegated to post deployment of an application
  - This also leads to reputational risk

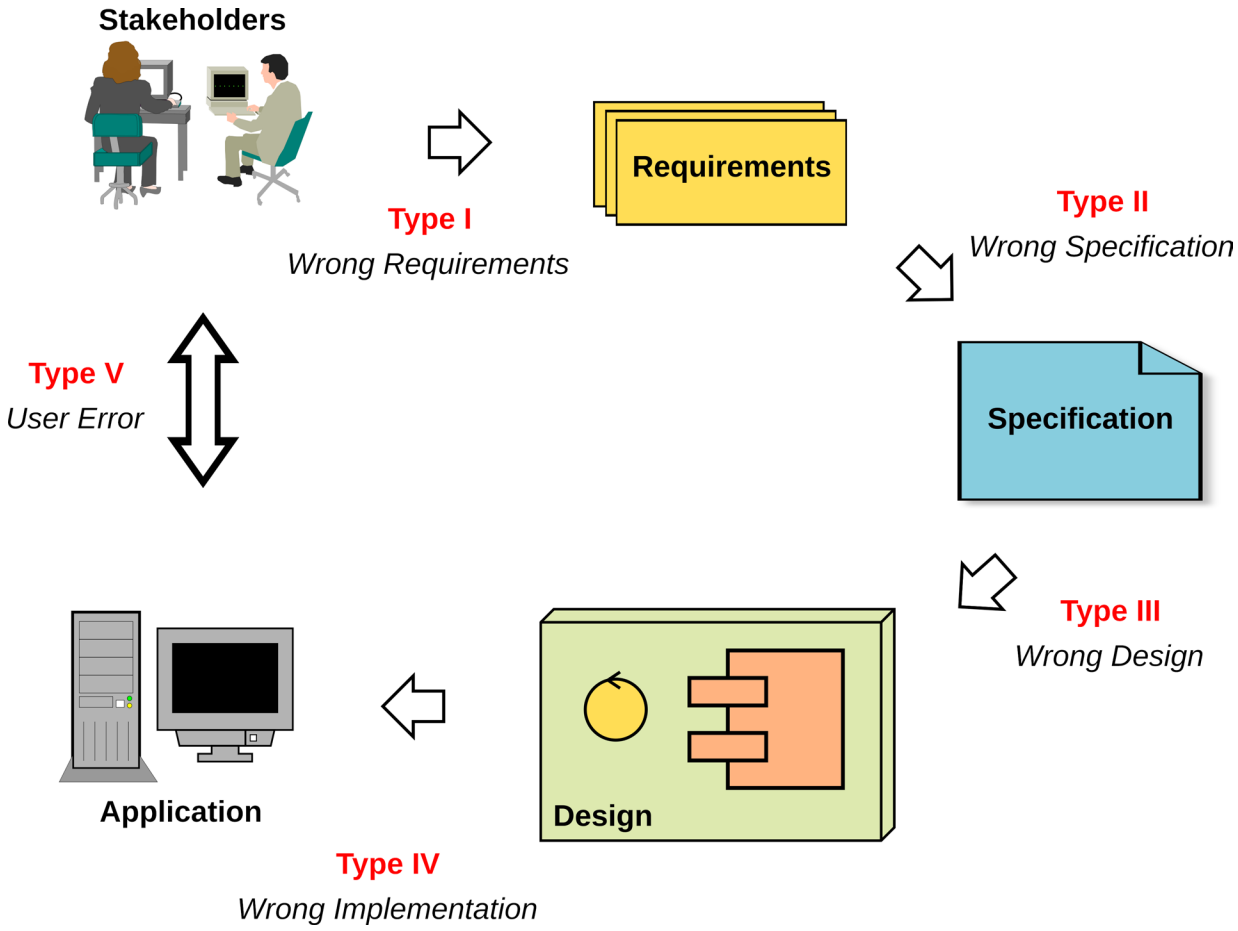
GENERIC DEVELOPMENT PROCESS

- Most of the standard representations of the generic software development process are similar to the diagram
  - Traditionally, the deployment and retirement phases are ignored
  - DevOps does rectify this to a degree
  - Even Agile methodologies often only deal with the phases shown, not the operations and retirement phases



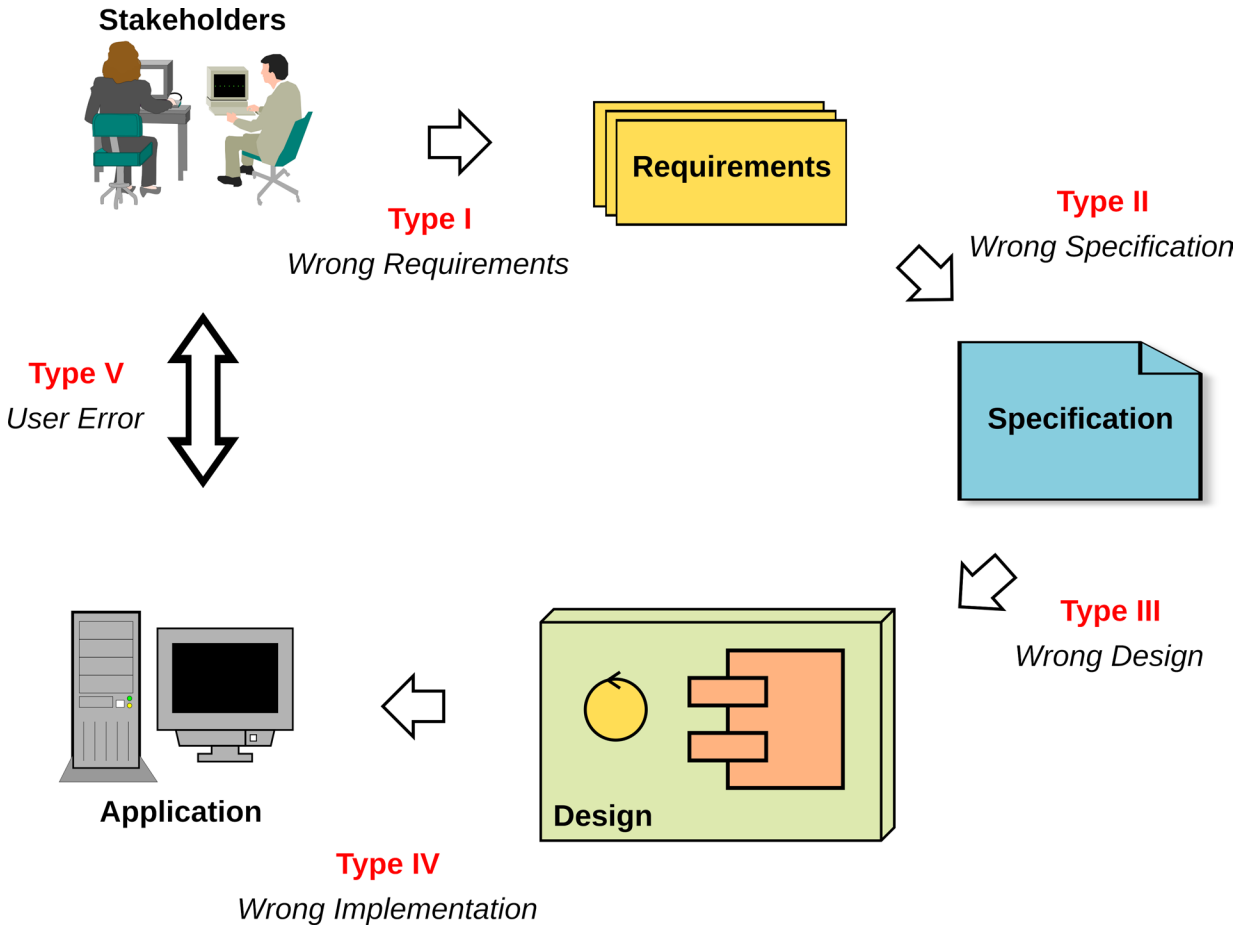
WHERE THINGS GO WRONG

- Type I: Requirements are wrong
  - This is the most difficult kind of error to correct since it cannot be detected within the project but can only be detected with the participation of the stakeholders who are outside the development effort
  - In the worst case scenario, identification of the error takes place after the finished product is deployed and being used by the stakeholders



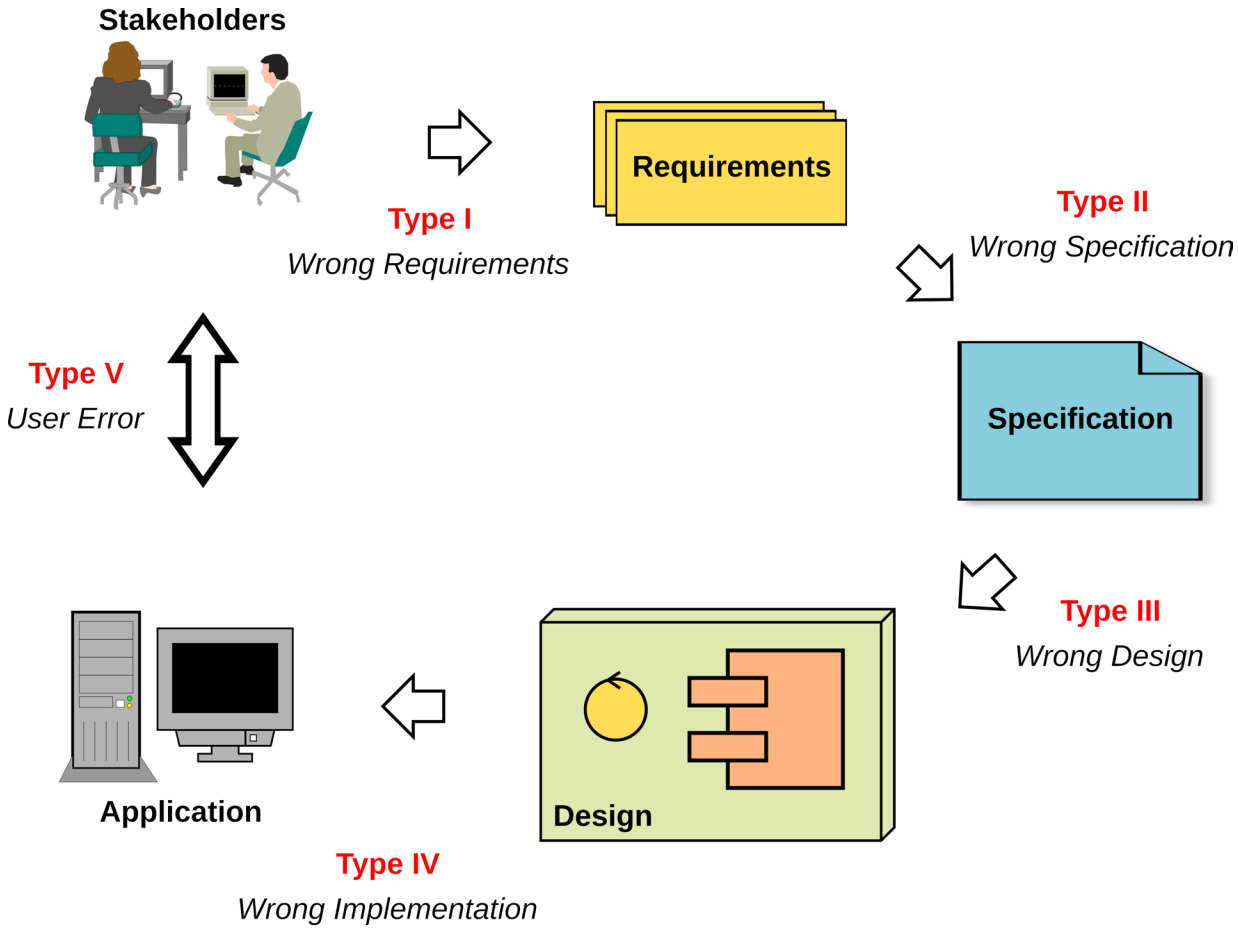
WHERE THINGS GO WRONG

- Type II: Specifications do not satisfy the requirements
  - Our system may work perfectly, but it does the wrong thing perfectly
- A type II error occurs if there are:
  - Requirements that are not addressed by the specification (omissions)
  - Features providing functionality that does not meet any requirement (surprise functionality)
  - Features that do not satisfy the requirements correctly (failures)



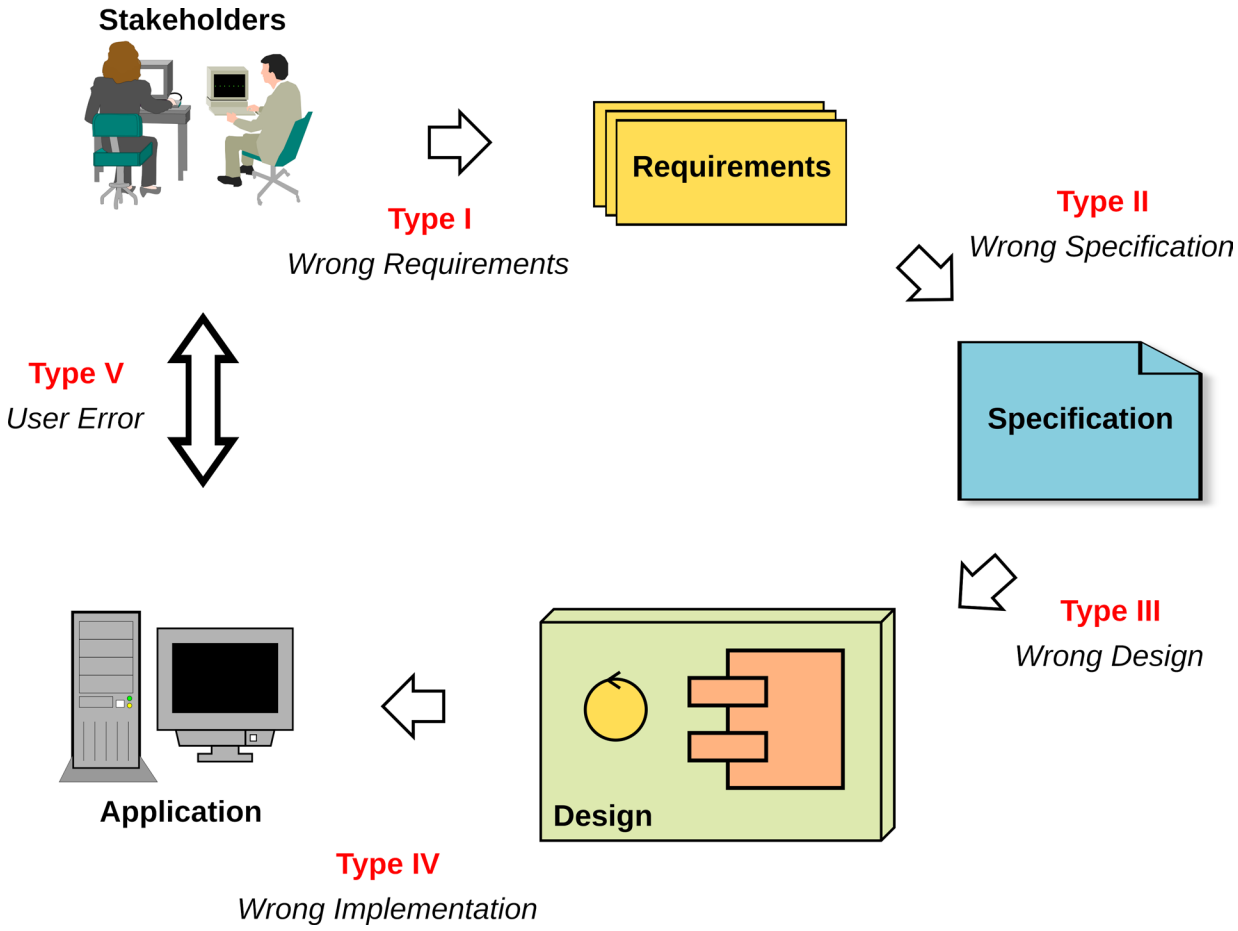
WHERE THINGS GO WRONG

- Type III: System design does not meet the specifications
  - Following the design leads to a system that does not operate as specified
  - For example, the specifications may require the use of encryption for security, but the encryption methodologies were omitted in the design by mistake
  - Or the architecture proposed cannot handle the projected throughput, or exceeds the allowable response time



WHERE THINGS GO WRONG

- Type IV: The system build and artifacts do not conform to the design
  - Someone made mistakes in building the system
  - Whether writing code or performing some other construction task that were specified in the design
  - Omitting error handlers for example

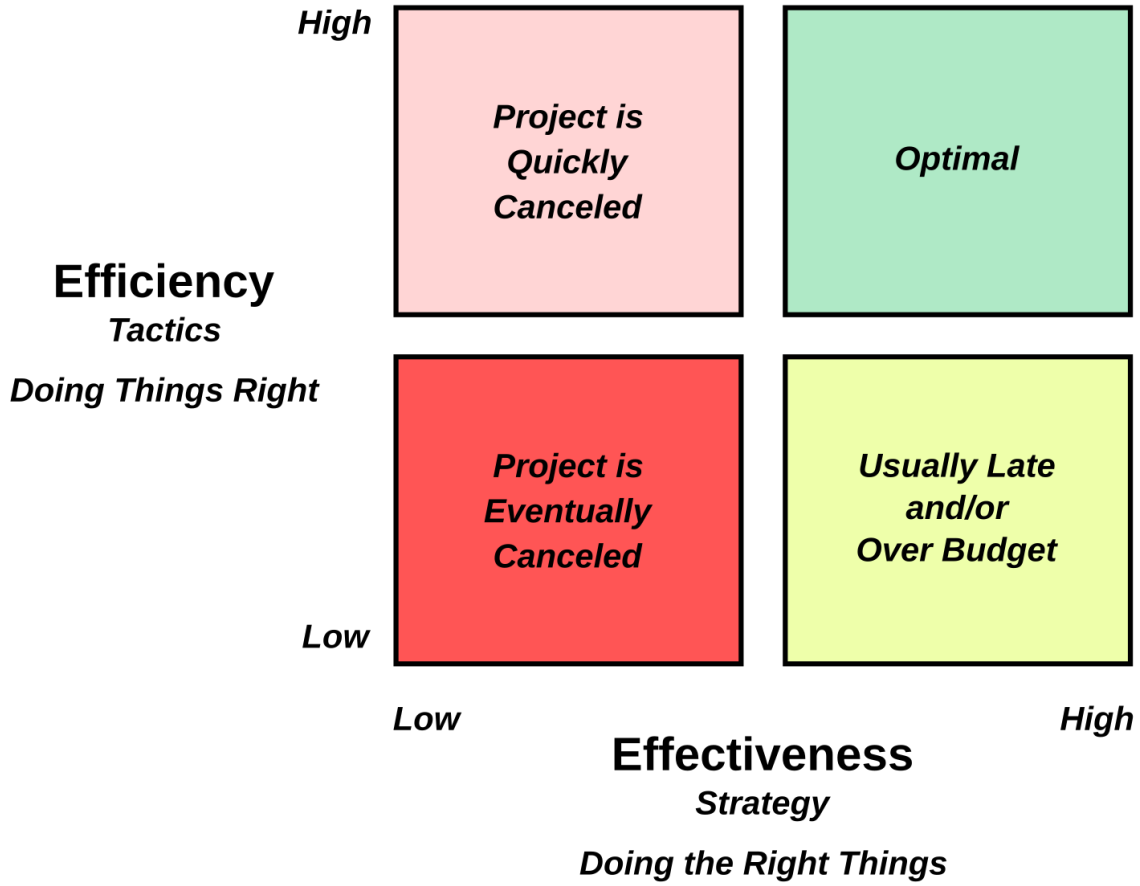


WHERE THINGS GO WRONG

- Type V: User Error
- This type of error is an unavoidable human error
  - The problem is that people do not always behave rationally and, while we may have designed and implemented everything correctly, it may be used in an irrational manner
  - It can also happen because users don't know how to use the system or they are using it in a different context that it was intended
  - The study of this sort of predictable but irrational behavior is studied by the field of Behavioral Economics

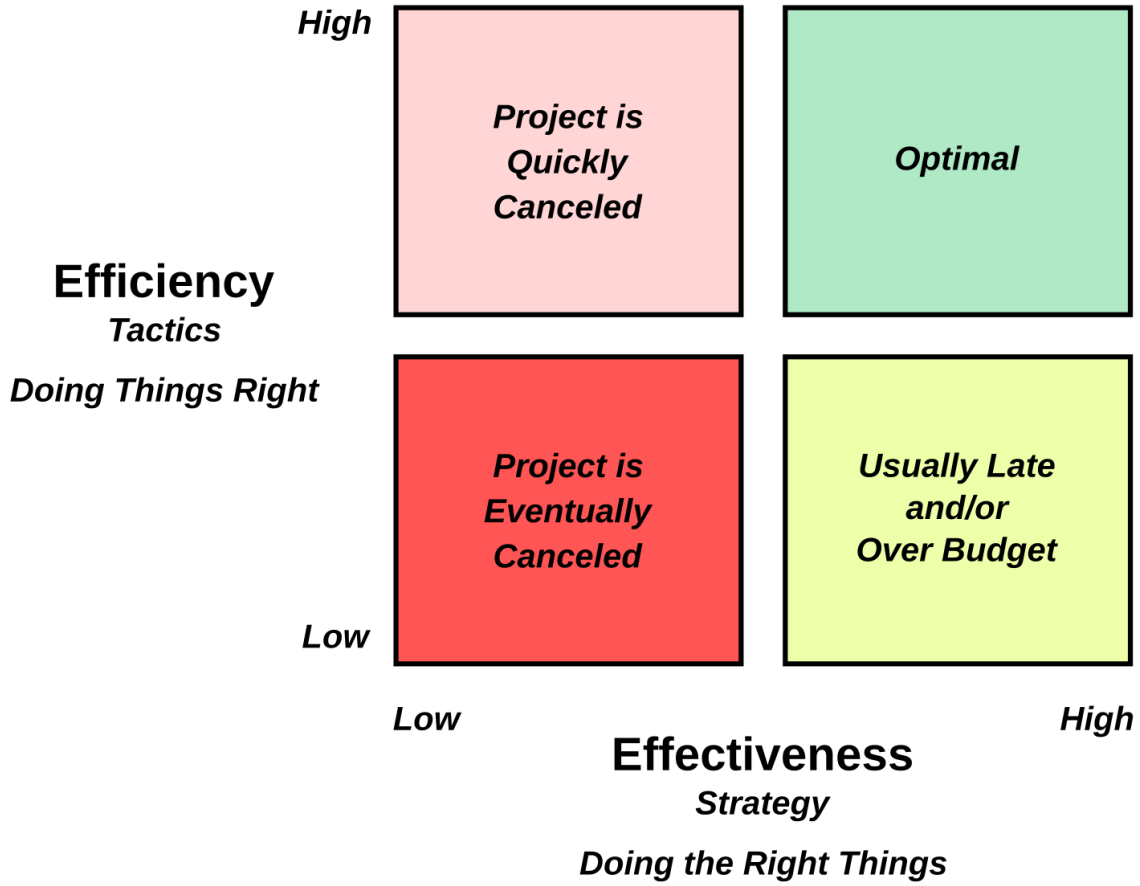
EFFICIENCY AND EFFECTIVENESS

- Effectiveness:
  - A process is effective when it produces the right result
  - We often call this delivering to spec
    - Delivering software that meets the stakeholder needs and solves the business problems as it was intended to
  - Effective software development processes build the right software



EFFICIENCY AND EFFECTIVENESS

- Efficiency:
  - A process is efficient when it is optimal in terms of how development resources are used
  - One of the main symptoms of a process inefficiency is how much rework is done during and after development
  - Rework is where previous work has to be done over because of errors made previously
  - Efficient software development processes build software correctly with the optimal use of resources

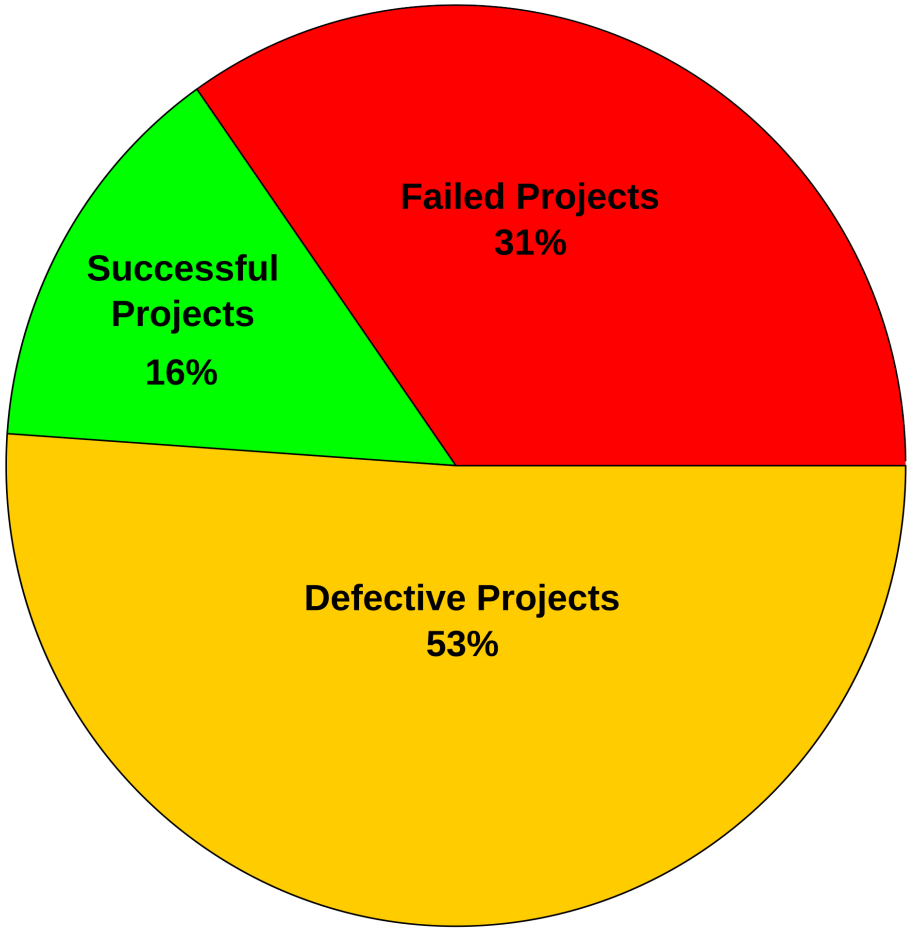


ERROR IMPACT

- These errors introduce risk and potential resilience failure
  - Unless the errors are caught, we have introduced risk into the product
  - We also have introduced multiple points of potential failure
  - If we unaware of where the potential point of failures are, resilience is compromised
    - *"The most important single aspect of software development is to be clear about what you are trying to build. Dijkstra "*
  - If we don't know what we are supposed to be building
    - Then we will never know if we are building the right thing
  - The data suggests that projects often don't know what they supposed to be building

THE CHAOS REPORT

- Annual report from the Standish group
  - Defines three resolutions for technical projects:
- Resolution Type 1 – Successful Projects:
  - The project is completed on-time and on-budget
  - All features and functions as initially specified.
  - Acceptance tests all pass
- Resolution Type 2 – Defective Projects:
  - The project is completed and operational but over-budget and/or late and/or is delivered fewer features and functions than originally specified
  - Usually with acceptance test failures
- Resolution Type 3 – Failed Projects:
  - The project is canceled at some point during the development cycle



SUCCESSFUL PROJECT FACTORS

- The table lists the factors that were identified by people on the project as being responsible for a project being successful
  - The red italicized factors are those that are related to requirements and interactions with the users and stakeholders
  - Two of the top three items concern quality of requirements and getting the users involved
  - The fifth item, realistic expectations, suggest that the developers had early input into the project to help the business understand what could be done within the time and budget constraints of the project

Success Factors	%
User Involvement	***16***
Executive Management Support	14
Clear Statement of Requirements	***13***
Proper Planning	***10***
Realistic Expectations	***8***
Smaller Project Milestones	8
Competent Staff	7
Ownership	5
Clear Vision and Objectives	***3***
Hard Working and Focused Staff	2
Other	13

DEFECTIVE PROJECT FACTORS

- The table lists the factors that were identified by people on the project as contributing to a project being challenged or defective
  - By definition a challenged project is one that was completed but either missed its budget targets, delivery date or did not perform as originally specified
  - The top three reasons why projects were defective or challenged have to do with specifications, requirements and user involvement
  - The more chaotic the requirements and specification, the less the developers know what they should be building

Defective Factors	%
Lack of User Involvement	***13***
Incomplete Requirements & Specs	***12***
Changing Requirements & Specs	***12***
Lack of Executive Support	8
Technology Incompetence	7
Lack of Resources	6
Unrealistic Expectations	***6***
Unclear Objectives	***5***
Unrealistic Time Frames	***4***
New Technology	4
Other	23

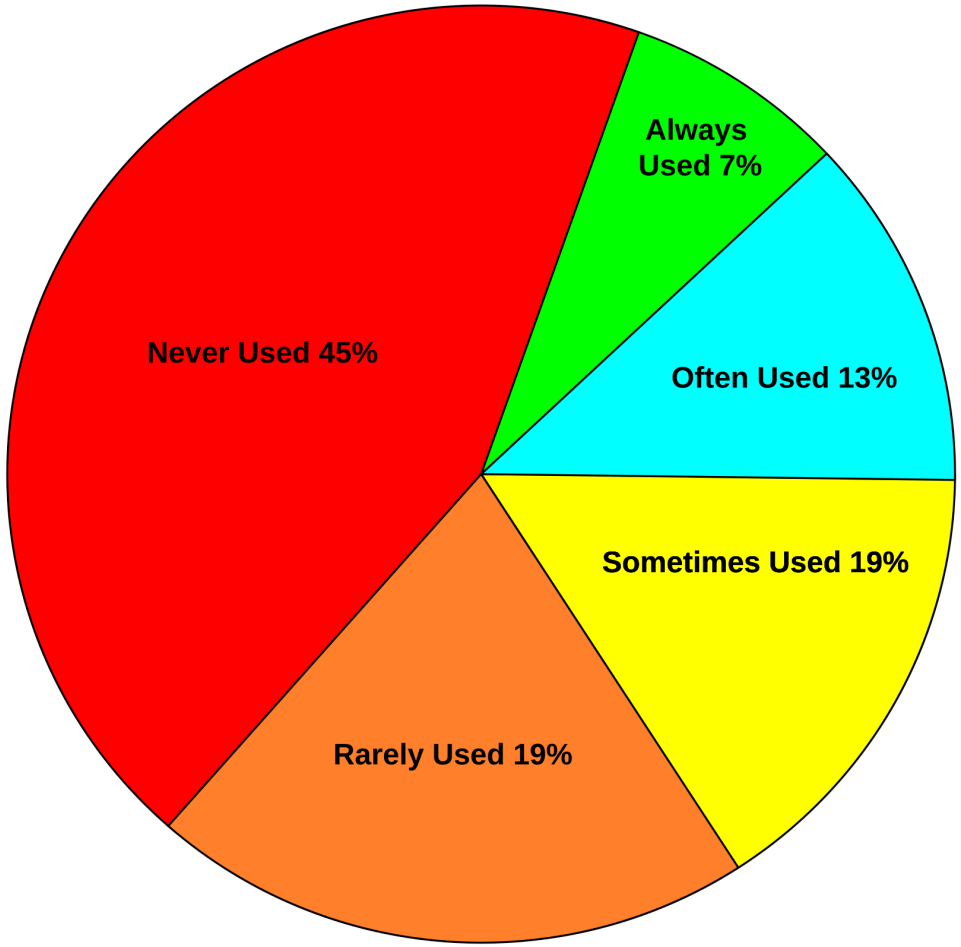
FAILED PROJECT FACTORS

- The table lists the factors that were identified as contributing projects being canceled as ranked by people on the project
  - Again, we see that requirements and specification issues and user involvement are right at the top
  - The unrealistic expectations suggests that what the business wanted to be built was beyond the technical solutions that could be reasonably be delivered by the developers

Failure Factors	%
Incomplete Requirements	***13***
Lack of User Involvement	***12***
Lack of Resources	11
Unrealistic Expectations	***10***
Changing Requirements & Specs	***9***
Lack of Management Support	9
Lack of Planning	***8***
Didn't Need It Any Longer	***8***
Lack of IT Management	6
Technological Illiteracy	4
Other	10

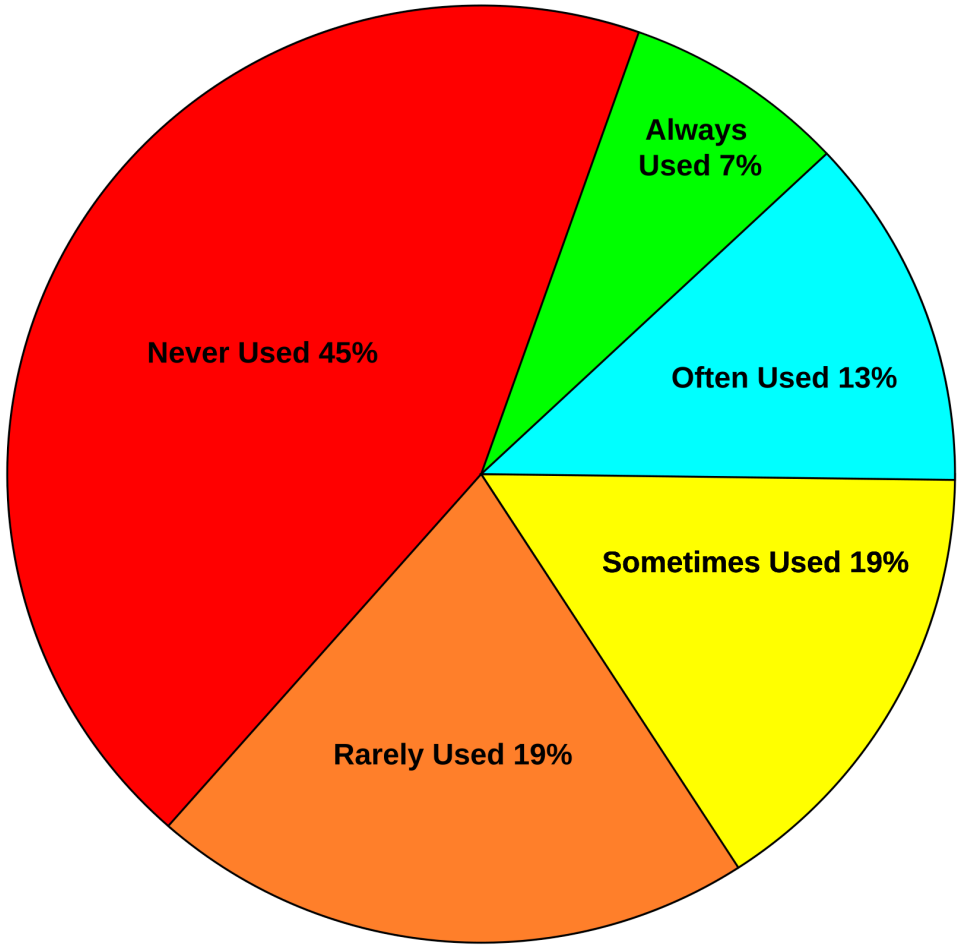
OVER ENGINEERING

- The diagram shows the frequency of use of the features in a delivered software project
  - These numbers are from the Chaos report
  - Only 20% of the delivered features are always or often used
  - Almost half the features are never used on average
- This means that about 45% of the development effort is wasted
  - We used resources and time to build things that never needed to be built
  - Usually happens because the feature choice is made by developers working without knowing the requirements the system needs to satisfy



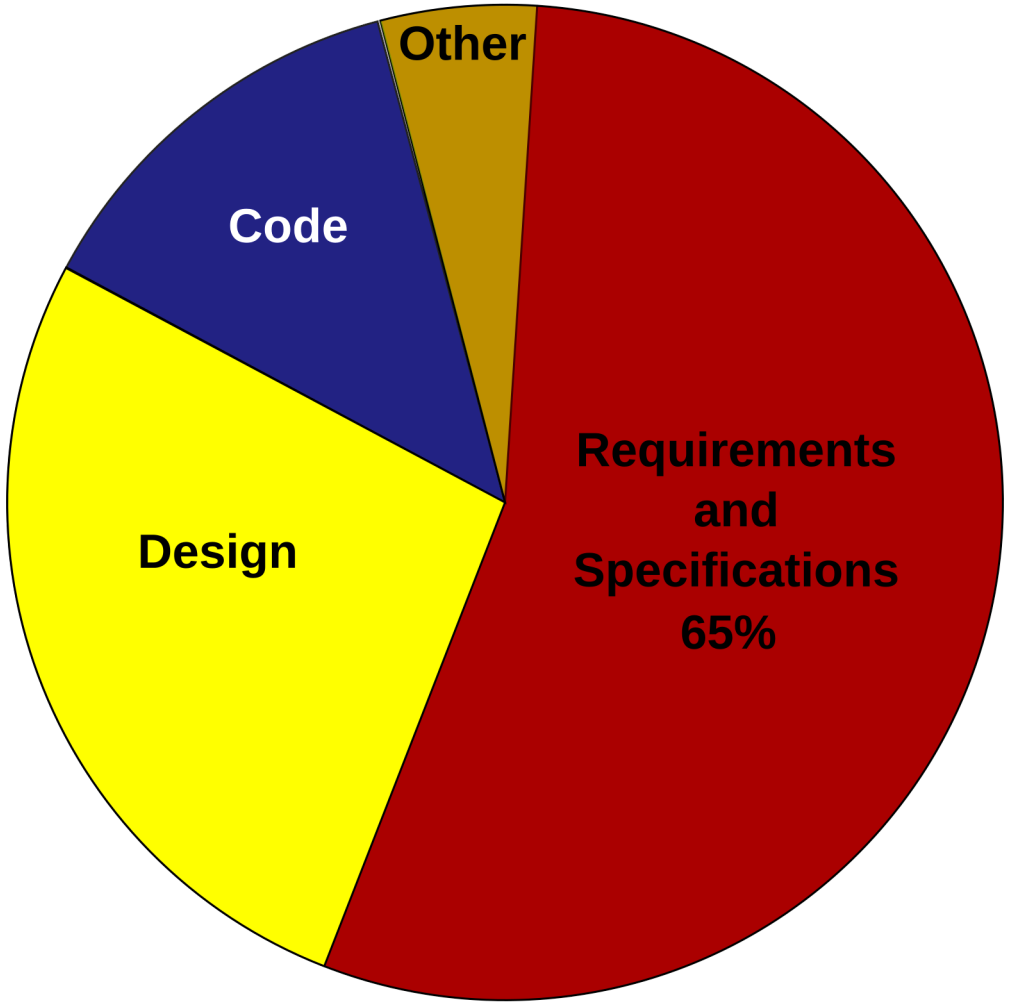
OVER ENGINEERING

- Over engineering is a combination of development ineffectiveness and inefficiency
  - Because we do not know what the end user really needs, we waste resources building things that never needed to be built at all
  - The application is ineffective in solving the original motivating need or requirement
  - In terms of efficiency, if we focus on the most important features first and defer building features that contribute little value, we have increased the efficiency of the process without compromising effectiveness
- Creates significant risk and resilience problems



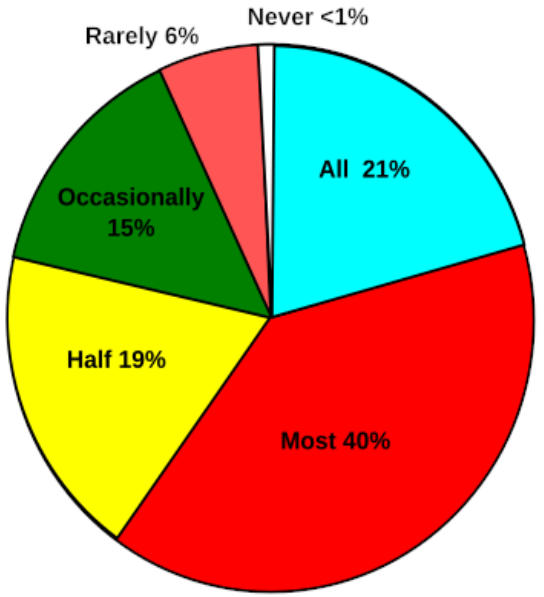
SOURCE OF ERRORS

- In a study by Gupta et al, the place in the process where errors originated from is shown in the pie chart
- What is the impact of errors in previous phases of a project?
  - We either get a defective or canceled project
  - Or we have to do a lot of rework to correct errors that have resulted working from wrong requirements or specification



REWORK EFFORT

- All of this so far can seem bit academic until we ask the really important
- Question: how much of developer time is taken up with fixing errors that could have been prevented?
  - 61% of the IT professionals surveyed said they spent all or most of their time fixing things they felt were preventable
  - When we add in those who spend half their time fixing mistakes, then 80% of IT professionals spend at least half their time on is fixing mistakes that could have been prevented



:

Geneca Survey of IT Professionals:

"How much of your time do you spend on avoidable or preventable rework (eg. changing requirements, missing features)?"

Results are consistent with other studies.

RISK AND RESILIENCE ISSUES

- Errors in the planning, design and building of software
  - Are a root cause of risk and resilience failures in other environments, like operations
  - The lack of controls in development forces more controls in operations
  - Could be considered inherited risk from development into operations
- Errors that propagate into operations means that
  - The resilience of the software is essentially unknown
  - Failures and outages blindside operations
  - There are no clear ways to return to operations

RISK AND RESILIENCE ISSUES

- Even if the software itself is error free
  - There are risks that
    - The business requirements for the software have changed
    - The operational environment the software runs in has changed
    - The regulatory requirements that govern the software functions have changed
  - The designed resilience features of the software
    - May not satisfy changed functional or operational requirements
    - May not work in a changed operations environment
    - May not meet changed recovery requirements

SCRUM

- Scrum was not originally intended for software development
  - Originally proposed in industrial manufacturing in the 1950s-1980s
  - Very influential: basis of Lean manufacturing, implemented in the Toyota production system
- In the 1980s, there was a major crisis in software development
  - Software was being developed in a big-bang approach
    - Siloed teams (design, development, testing) worked in isolation with one-time hand offs of artifacts
  - NATO software engineering conference in 1968 was held to address the high failure rate of these big-bang projects
    - The conclusion of this and other similar conferences was that in incremental and iterative approach, like Kaizen (continuous improvement) and lean approaches to product development were needed

SCRUM

- The term Scrum is first used in 1986 with reference to new product development
  - Hirotaka Takeuchi and Ikujiro Nonaka “The New New Product Development Game”
- Jeff Sutherland & Ken Schwaber (1997)
  - Presented the first public paper on Scrum: "SCRUM Development Process."
  - Emphasized:
    - Empirical process control theory (transparency, inspection, adaptation)
    - Lean principles
    - Nonaka & Takeuchi’s knowledge-creation theory (SECI) – the basis for cross functional teams
- The basic ideas expressed in Scrum started to be adopted generally
  - Primarily by teams working with volatile types of projects
  - Often smaller teams of developers working closely with the business side

SCRUM

- In the 1980s and 1990s
  - Companies experimented with using what they called adaptive development
  - IBM, DuPont, and others experimented with iterative prototyping and empirical process control
  - Characterized by
    - Use of successive prototypes to get feedback on requirements, design and performance
    - Short iterations (one month or less)
    - Cross-functional teams
    - Daily meetings for synchronization
    - A prioritized feature list

FORMALIZATION OF SCRUM

- 2002
  - Scrum Alliance founded to promote Scrum through certifications, communities, and training
- 2009
  - Jeff Sutherland & Ken Schwaber publish "Scrum Guide (1st Edition)"
  - First formal codification of Scrum as a lightweight framework.
    - Defined its roles, events, artifacts, and rules
- 2010s
  - Widespread adoption beyond software
    - Scrum principles applied to: Marketing, HR, education, and hardware development
    - Applied to entire organizations (Enterprise Scrum, Scrum@Scale, LeSS, Nexus)

FORMALIZATION OF SCRUM

- 2010
  - Scrum.org founded by Ken Schwaber
    - Alternative certifying body with a focus on assessment-driven learning
- 2017
  - Major scrum guide update
    - More emphasis on values: commitment, focus, openness, respect, courage
- 2020
  - Scrum guide simplification
    - Roles streamlined
    - Focus on a single Scrum Team with a shared goal
    - Removed prescriptive elements to broaden use outside software

MAJOR IDEAS INCORPORATED INTO SCRUM

- Empirical process control
  - Transparency, inspection, adaptation
  - From Lean manufacturing and scientific method
- Cross-functional, self-organizing teams
  - From Takeuchi & Nonaka and Lean
- Iterative and incremental development
  - Inspired by early software engineering pioneers

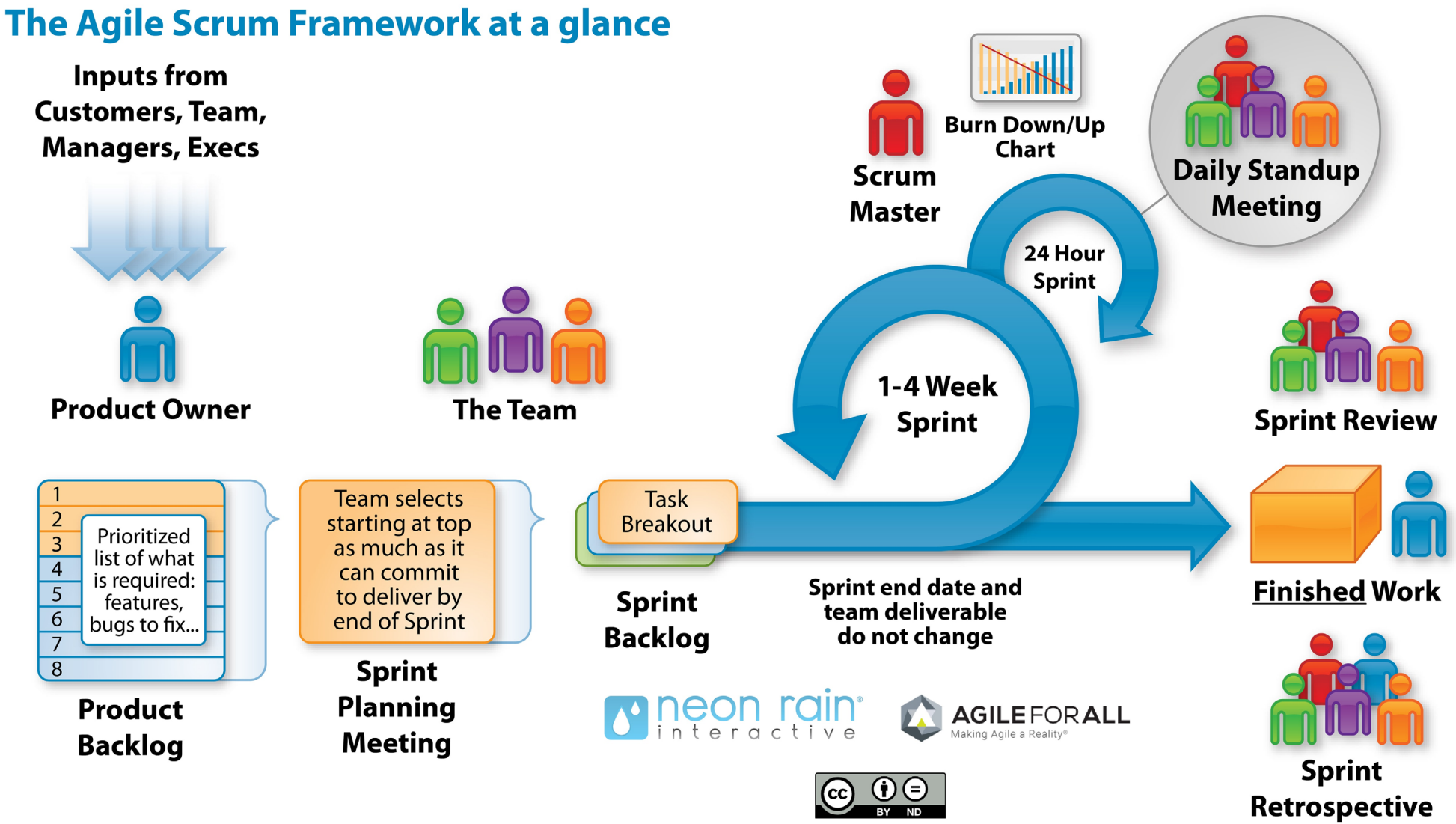
MAJOR IDEAS INCORPORATED INTO SCRUM

- Small batches and feedback loops
  - From Lean manufacturing and prototyping
- Knowledge creation and tacit knowledge sharing
  - From Nonaka's work on organizational learning
- Focus on people and communication
  - From Fred Brooks and Extreme Programming values
- Timeboxing
  - Borrowed from project management and software engineering experiments in the 1980s–90s

FRAMEWORKS AND METHODOLOGIES

- A common mistake is to call Scrum an Agile methodology
  - It is not a methodology, it is a process framework
  - Failure to have a methodology defined while using Scrum negates the value of using it
- Scrum is process-focused
  - Managing what and when
- Agile methodologies are practice-focused
  - Describing the concrete steps to get the work
- The two complement each other
  - Scrum without a software engineering methodology risks low quality
  - A software engineering methodology without Scrum risks lack of direction and prioritization

SCRUM AT A GLANCE



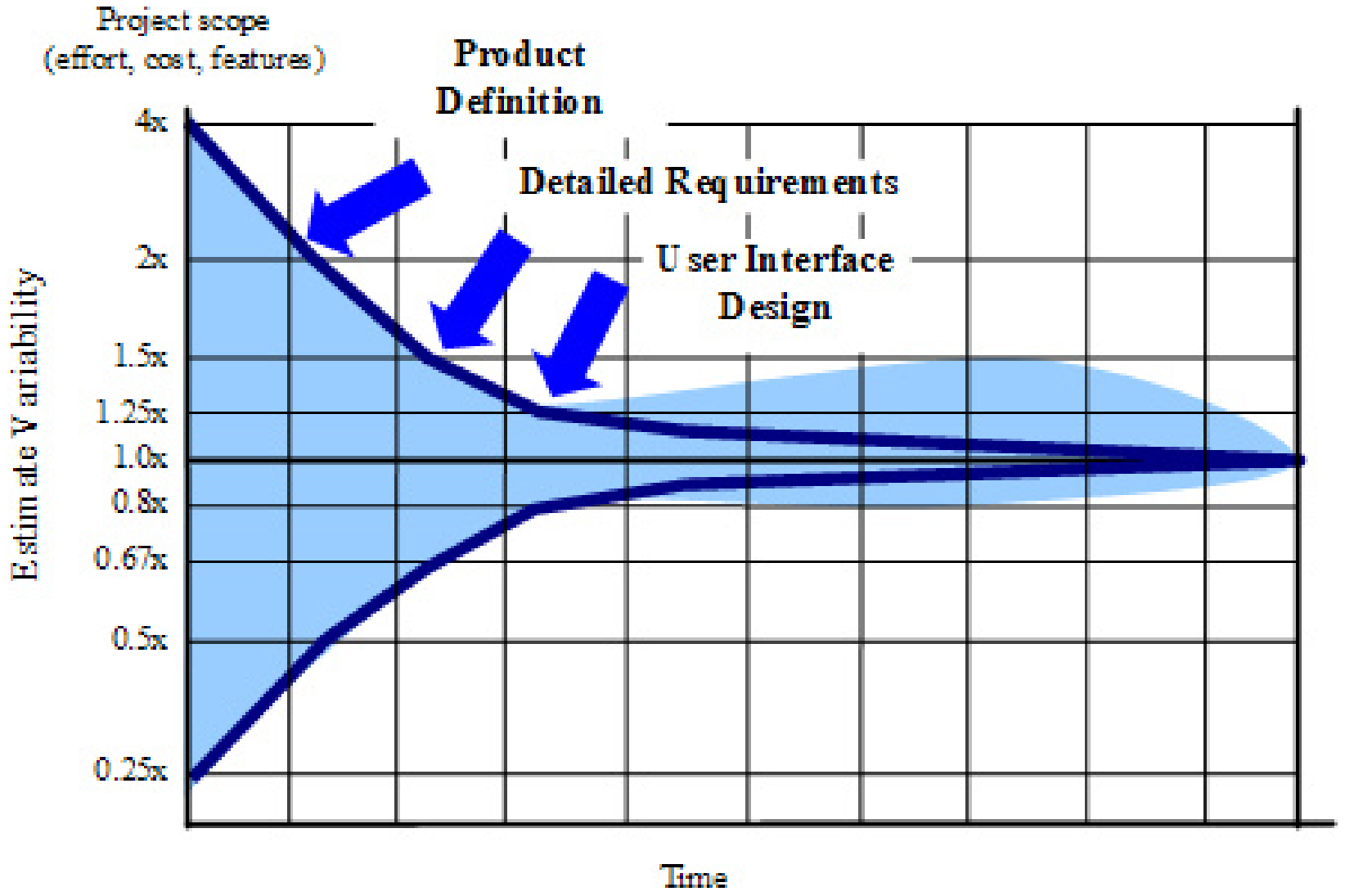
UNCERTAINTY

- A big bang process is not generally possible unless
  - Everything is known about the requirements, possible solutions, designs
    - All and any potential causes of failure for the application
    - Very common for stable engineering environments like civil engineering
    - Also used for very high risk of mission critical software (eg, nuclear reactor controls)
  - However, this is not the majority of product development projects
    - We may not know all the requirements until users see a prototype
    - We may not know if proposed solutions actually solve the problem until we prototype
    - We may not know if designs are feasible until we try them out (reliable, affordable and resilient)
  - These unknowns at each stage of the process create “slippage”
    - Unless resolved, typically the project fails or is defective

THE CONE OF UNCERTAINTY

- Comes from the challenge of software estimation
  - Goal is to establish estimates of costs, timelines and functionality
- Estimates are based on available data
  - Early in the project, the data may only be available as best guesses
  - This means estimates have a range of uncertainty
- As the project progresses
  - The estimates can be refined with data collected during development
  - This narrows the range of the original estimates
  - At the end of the project, the estimate converges to the actual final values
  - These may be out of the range of the original estimates

THE CONE OF UNCERTAINTY

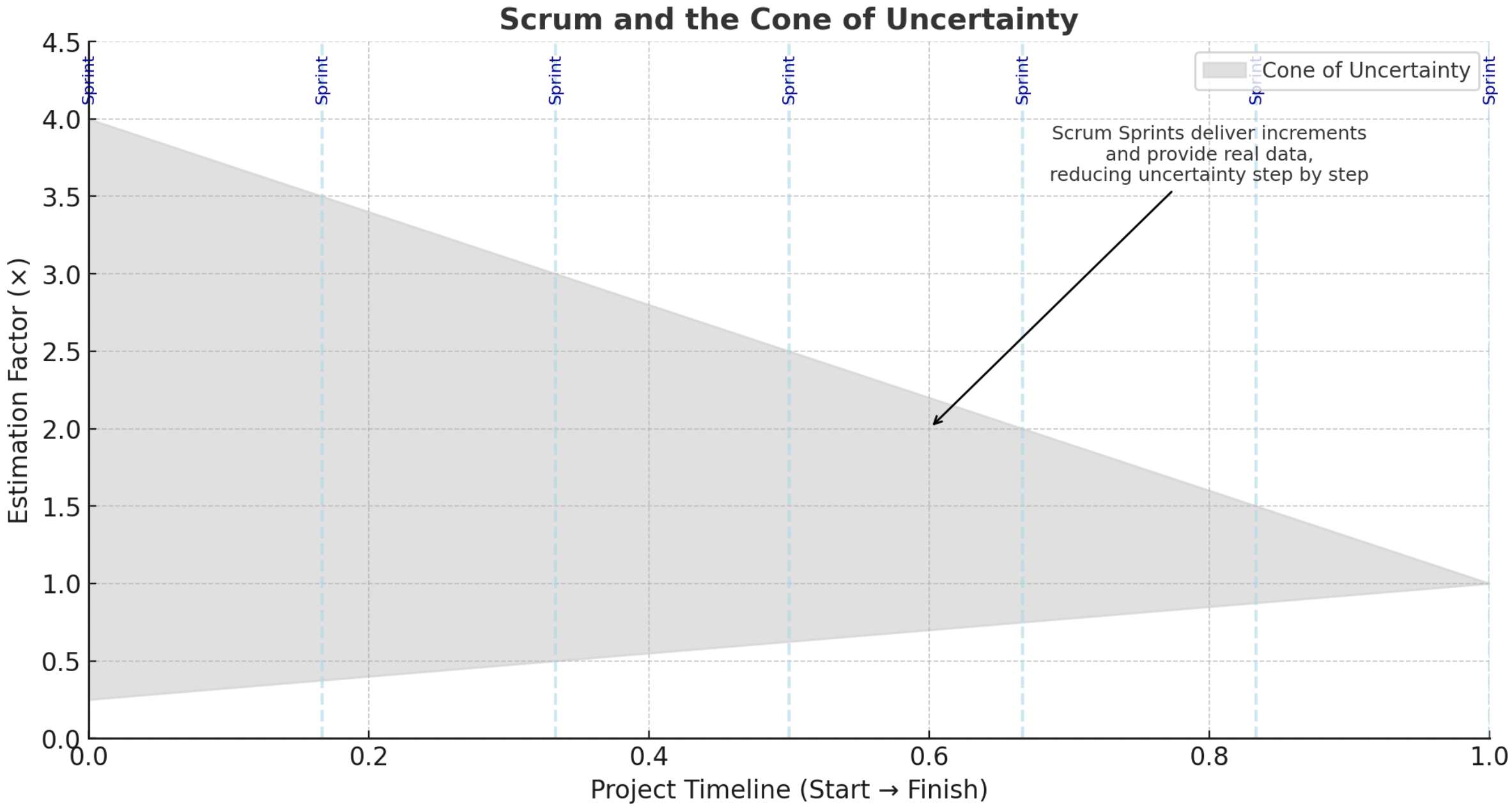


SCRUM AS A RISK MANAGEMENT FRAMEWORK

- Scrum is inherently designed to manage risk
  - It integrates well with risk practices
  - Even if it doesn't use traditional risk registers or ISO-style controls
- Risk related features
  - Short, iterative cycles (sprints) limits exposure to large failures
  - Continuous feedback and review: identify problems early
  - Empowered teams and transparency: enable quick response to uncertainty
- Scrum is risk-responsive by design
  - Excellent foundation for integrating formal risk and resilience practices

SCRUM AND UNCERTAINTY

- The cone of uncertainty
  - Warns us that early estimates are unreliable, including risk estimates
  - Especially the estimates remain fixed during the development process
- Scrum approaches this problem by using
  - Iterative delivery to reduce uncertainty through working increments
  - Empirical velocity-based forecasting using evidence, to make estimates more accurate
  - Backlog reprioritization to optimize value as the cone narrows
- Scrum actively forces the cone to narrow
  - Studies on IT projects shows that incremental and iterative processes have lower rates of project failure and have more accurate estimates of timelines and budgets



RISK MANAGEMENT IN SCRUM

Scrum Element	Risk Interaction	Example
Product Backlog	Risks can be logged as backlog items or acceptance criteria. Prioritization considers risk impact.	"Mitigate data loss by implementing backup function" as a backlog item.
Sprint Planning	Teams identify technical and delivery risks for upcoming sprint.	Team notes dependency on external API as a risk.
Daily Scrum	Ongoing monitoring of impediments and risks.	"The vendor API was delayed — risk of missing sprint goal."
Sprint Review	Early exposure to customer feedback reduces risk of product misalignment.	Product Owner and stakeholders test increment and uncover usability issues.
Sprint Retrospective	Focus on process-level risks and resilience improvements.	"We need better version control to reduce deployment errors."

FORMAL RISK MANAGEMENT IN SCRUM

For organizations that must comply with risk frameworks (e.g., ISO 31000, NIST, or operational risk policies), Scrum teams can **embed lightweight risk processes** that map naturally to Scrum events:

Risk Activity	Scrum Integration
Risk Identification	During backlog refinement and sprint planning.
Risk Assessment (Impact × Likelihood)	Conduct informally using team discussion or a lightweight matrix.
Risk Mitigation Planning	Add risk responses as backlog tasks or "spikes."
Risk Monitoring	Handled daily in standups and through burn-down charts.
Risk Reporting	Communicated during sprint reviews or retrospectives.

SCRUM AND RESILIENCE

- Iterative delivery
  - Builds system redundancy and recovery capability
- Scrum uses short sprints
  - Ensure that software and systems are developed, tested, and deployed in small, recoverable increments which supports early identification of potential points of failure
- Improves resilience by having
  - *Smaller blast radius:* If a deployment fails, only a limited set of features are affected
  - *Frequent integration/testing:* Continuous validation ensures the system is always in a working state
  - *Regular delivery of value:* If a disruption occurs, previously delivered increments remain usable and stable, and reduces time to recovery
  - Example:
    - A financial services platform using Scrum deploys new modules every two weeks
    - When a database migration issue arises, rollback is simple: only the sprint's incremental update needs to be reverted, not the entire system

SCRUM AND RESILIENCE

- Embedded inspection and adaptation
  - Empirical process control: inspect–adapt–transparency
  - These feedback loops create an organizational culture where failure is expected, analyzed, and used to improve recovery mechanisms for the products

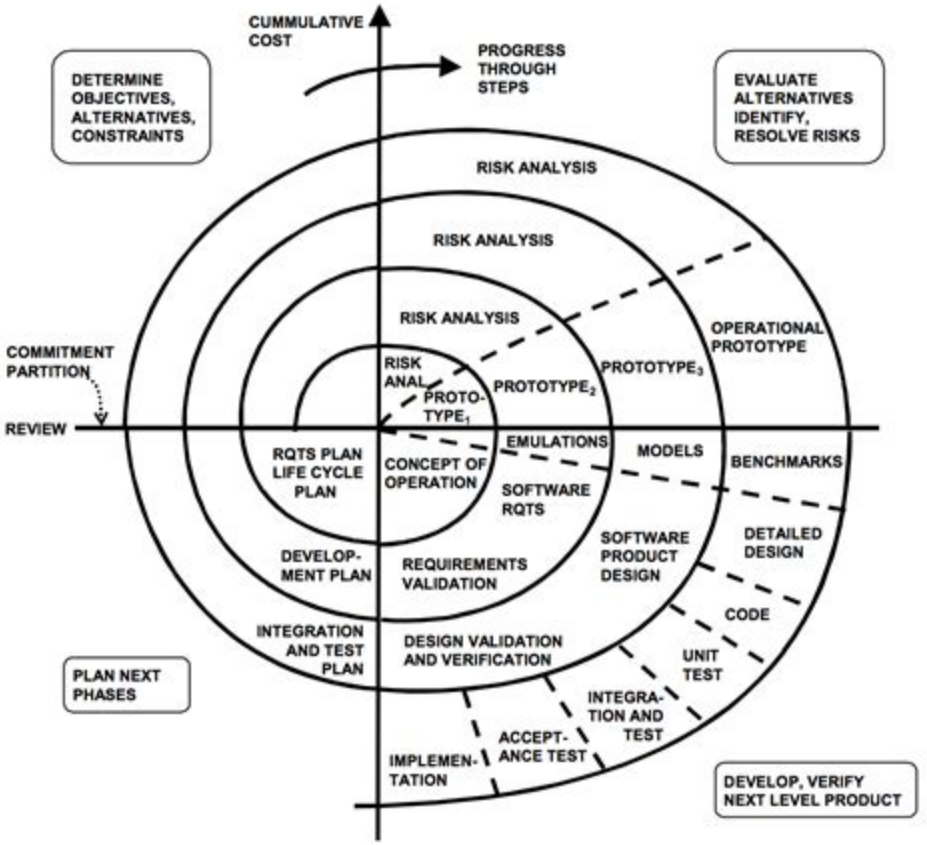
Scrum Mechanism	Resilience Impact
Daily Scrum	Early detection of incidents, failed builds, or infrastructure instability.
Sprint Review	Exposes system weaknesses and performance gaps to stakeholders regularly.
Sprint Retrospective	Drives systemic learning — e.g., identifying root causes of outages or recovery delays.

SCRUM AND RESILIENCE

- Cross-functional teams reduce single points of failure
  - Naturally distributes knowledge and responsibility across roles
  - Reduces organizational fragility
    - Developers, testers, and ops staff collaborate daily: no “handoff silos”
    - Knowledge sharing collaborative processes ensures redundancy in expertise
    - Avoids having only one group or individual who understands the system resilience
  - Resilience knowledge about the application
    - Becomes an emergent property of how the team operates
    - Not something added on later

SPIRAL METHODOLOGY

- Developed by Barry Boehm in 1986
  - Spiral Model integrates elements of both design and prototyping-in-stages
  - Introduced iterative development into the waterfall approach
- Emphasizes risk-driven development.
  - Each iteration (or “spiral”) is designed to identify and reduce key risks early
  - Including technical, financial, operational organizational
  - Before committing to the next phase



PHASES OF EACH SPIRAL

- Each cycle follows four repeating stages
  - Objective setting and planning
    - Define goals, constraints, and alternatives for the iteration
    - Determine what risks could impact progress
  - Risk analysis and evaluation
    - Identify and assess risks in technical, operational, or business dimensions
    - Develop mitigation strategies or prototypes to address them
  - Development and validation
    - Build and test deliverables (could be a prototype, a process change, or an operational control)
    - Incorporate feedback and validate against objectives
  - Review and planning for next iteration
    - Evaluate results and decide whether to continue, adjust, or terminate the project
    - Plan the next spiral with updated risk insights

SPIRAL METHODOLOGY

- Key characteristics
  - *Iterative*: Continuous refinement rather than a one-time plan
  - *Risk-driven*: Every cycle begins with risk assessment and mitigation
  - *Flexible*: Supports changes as understanding deepens
  - *Stakeholder Involvement*: Users and sponsors participate in reviewing progress and validating outcomes
- Relevance to risk and resilience
  - Models risk awareness as a process, not a one-time assessment
  - Encourages continuous improvement through iteration, similar to Agile retrospectives
  - Aligns well with resilience thinking: learn, adapt, and improve after each cycle
  - Provides a structured yet adaptive approach to managing uncertainty

SCRUM AND AGILE

- Agile was the general term adopted in 2001
  - Defined by owners of adaptive methodologies that shared a similar approach to development
  - Derived many of their ideas from their use of Scrum ideas
    - For example: Extreme Programming, Feature Driven Development
- Many of the features of Scrum were incorporated into the Agile Manifesto and the Agile Principles
  - Note that Scrum is NOT an Agile methodology, but its concepts were shared among Agile methodologies
  - Specifically:
    - Individuals and interactions
    - Working software prototypes
    - Customer collaboration and feedback loops
    - Responding to change in a planned systematic way

AGILE MANIFESTO

- Agile methodologies tend to incorporate Scrum practices
  - Scrum is focused on process
  - Agile tends to focus on the type of work done during the process
- Agile manifesto and principles
  - Summary of the common approach taken by the different Agile methodologies
  - The principles are a set of characteristics common to Agile teams
  - Each methodology has its own way of implementing the principles

AGILE PRINCIPLES

- Agile principles
  - Our highest priority is to satisfy the customer through early and continuous delivery of valuable software
  - Welcome changing requirements, even late in development
    - Agile processes harness change for the customer's competitive advantage
  - Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale
  - Business people and developers must work together daily throughout the project
  - Build projects around motivated individuals
    - Give them the environment and support they need, and trust them to get the job done
  - The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
  - Working software is the primary measure of progress

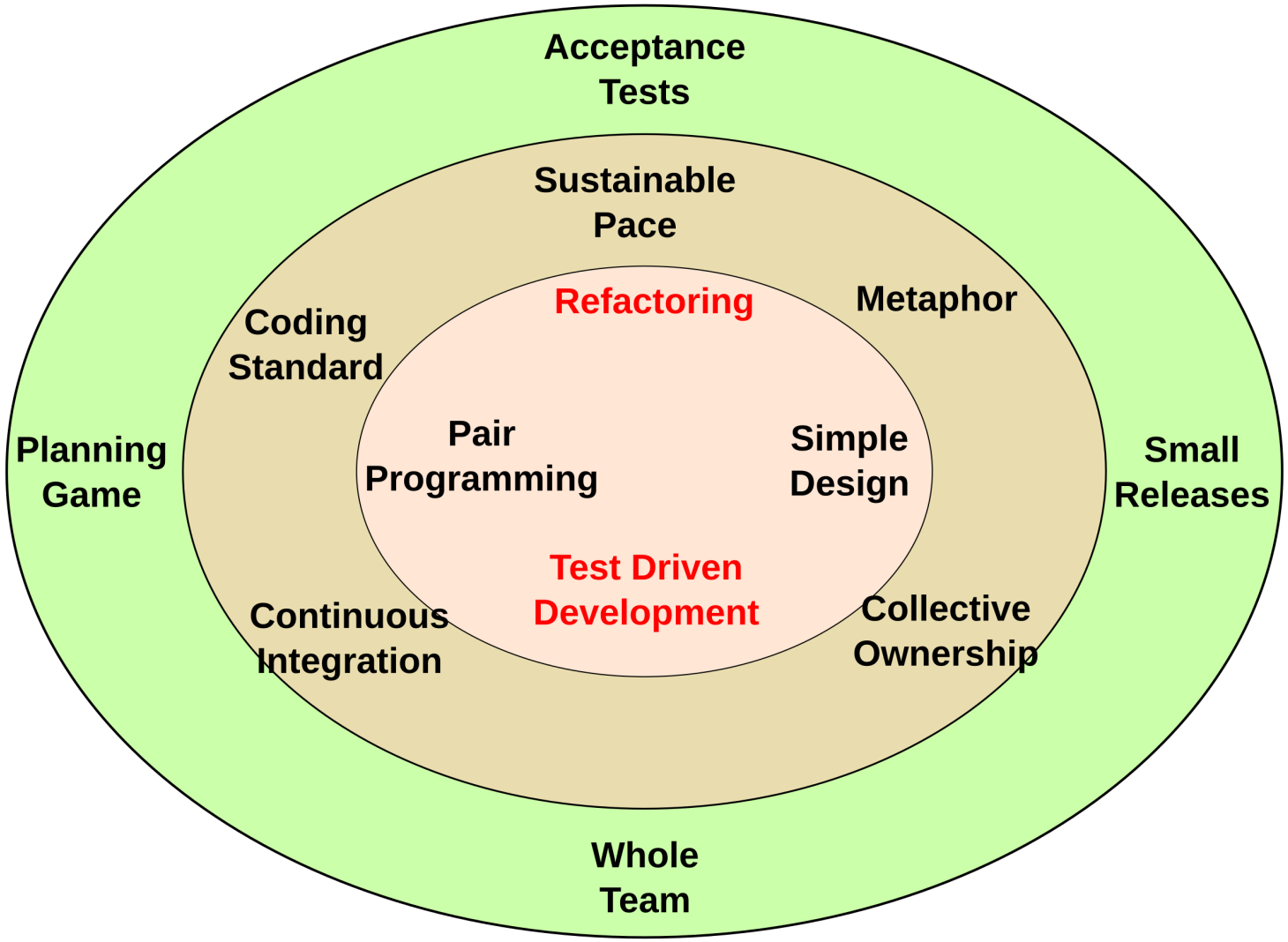
AGILE PRINCIPLES

- Agile processes promote sustainable development
  - The sponsors, developers, and users should be able to maintain a constant pace indefinitely
- Continuous attention to technical excellence and good design enhances agility
- Simplicity – the art of maximizing the amount of work not done – is essential
- The best architectures, requirements, and designs emerge from self-organizing teams
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly

EXTREME PROGRAMMING (XP)

- Developed in the late 1990s by Kent Beck
  - One of the original Agile methodologies
  - Designed to improve software quality and responsiveness to changing customer requirements through frequent releases, continuous feedback, and disciplined technical practices
- Core ideas
  - XP focuses on adaptability, collaboration, and technical excellence
  - Pushes core agile principles to the “extreme”
  - Ensuring that teams can deliver value rapidly and sustainably even in high-uncertainty environments

EXTREME PROGRAMMING (XP)



EXTREME PROGRAMMING (XP)

- Core practices
  - The Planning Game
    - Customers and developers cooperate to produce the maximum business value as rapidly as possible
  - Small Releases
    - Start with the smallest useful feature set.
    - Release early and often
    - Adding a few features each time
  - Metaphor
    - Each project has an organizing metaphor, to provide an easy to remember naming convention
  - Simple Design
    - Always use the simplest possible design that gets the job done
    - The requirements will change tomorrow, so only do what's needed to meet today's requirements
  - Continuous Testing (TDD):
    - Tests are written for a feature to be added, then the code is written
    - When the suite passes, the job is done

EXTREME PROGRAMMING (XP)

- Refactoring
  - Refactor out any duplicate code generated in a coding session
  - You can do this with confidence that you didn't break anything because you have the tests
- Pair Programming
  - All production code is written by two programmers sitting at one machine
  - Essentially, all code is reviewed as it is written
- Collective Code Ownership
  - No one "owns" a module
  - Anyone is expected to be able to work on any part of the codebase at any time
- Continuous Integration
  - All changes are integrated into the codebase at least daily
  - The tests have to run 100% both before and after integration

EXTREME PROGRAMMING (XP)

- 40Hour Work Week (Sustainable Pace)
  - Programmers go home on time, up to one week of overtime is allowed
- Onsite Customer Tests
  - Development team has continuous access to a real live customer
  - Someone who will actually be using the system who does continuous test generation and acceptance
- Coding Standards
  - Everyone codes to the same standards so that there is no way to tell by inspection who on the team worked on a specific piece of code

EXTREME PROGRAMMING (XP)

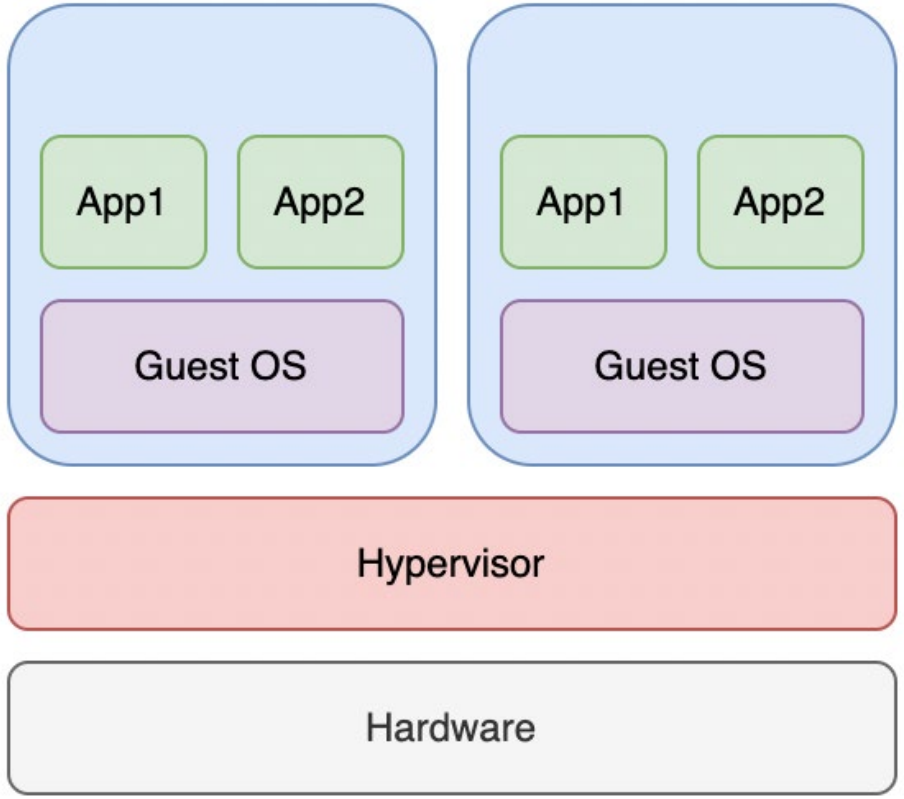
- Key characteristics:
  - *Highly iterative and incremental*: Frequent releases and feedback loops
  - *Human-centered*: Collaboration and trust are central
  - *Quality-Driven:* Testing, integration, and refactoring prevent defects
  - *Adaptable:* Embraces change as part of the process, not a disruption
- Relevance to risk and resilience:
  - *Risk reduction through immediacy:* Frequent integration and testing surface risks early
  - *Operational resilience:* Collective ownership builds fault tolerance in teams
  - *Process resilience:* Practices like refactoring and TDD create systems that can evolve safely
  - *Human resilience:* Sustainable pace and teamwork mitigate burnout and turnover risks

DEVOPS

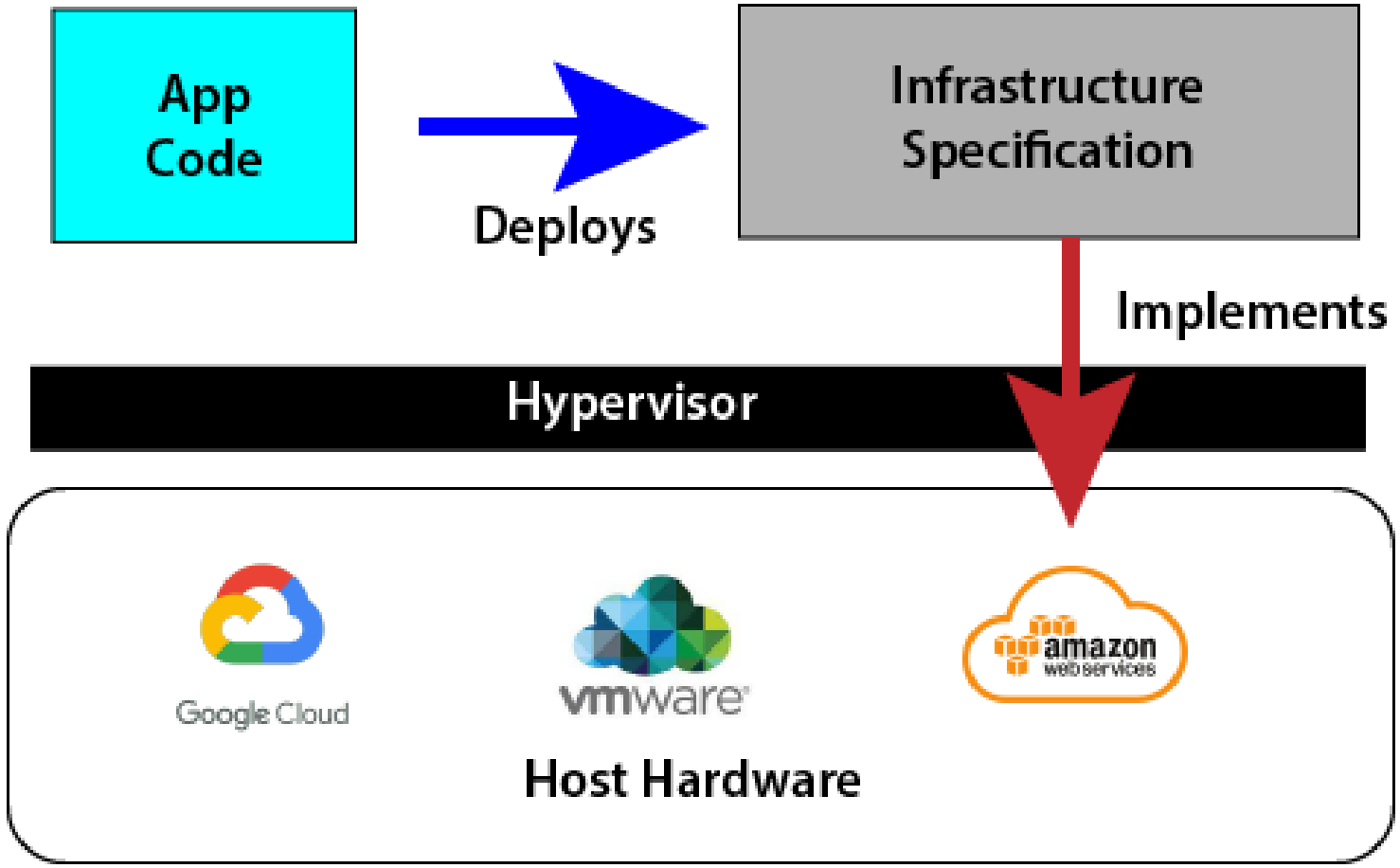
- The result of improvements in hardware speed and capacity
  - Previously the limiting factor in computing was hardware
- Faster and more powerful hardware
  - Meant that software did not fully utilize the hardware capability
  - The model of one operating system to one physical host became wasteful
- Virtualization
  - Developed by IBM in the 1970s to support multiple operating systems on a single physical host, at that time, mainframe
  - Established a translation layer between the virtual Os and real OS called a hypervisor

DEVOPS

- The virtual environment
  - Executes code on the hypervisor
  - Creates a virtual representation of a physical device like a computer or disk drive
  - Maps the virtual device to some underlying hardware resource on the underlying host
- Tools like Terraform are
  - Used to specify the virtual resources required
  - Executed by the hypervisor or equivalents
  - Called “Infrastructure as Code” or IoC
  - Note: this is a simplification of what really happens

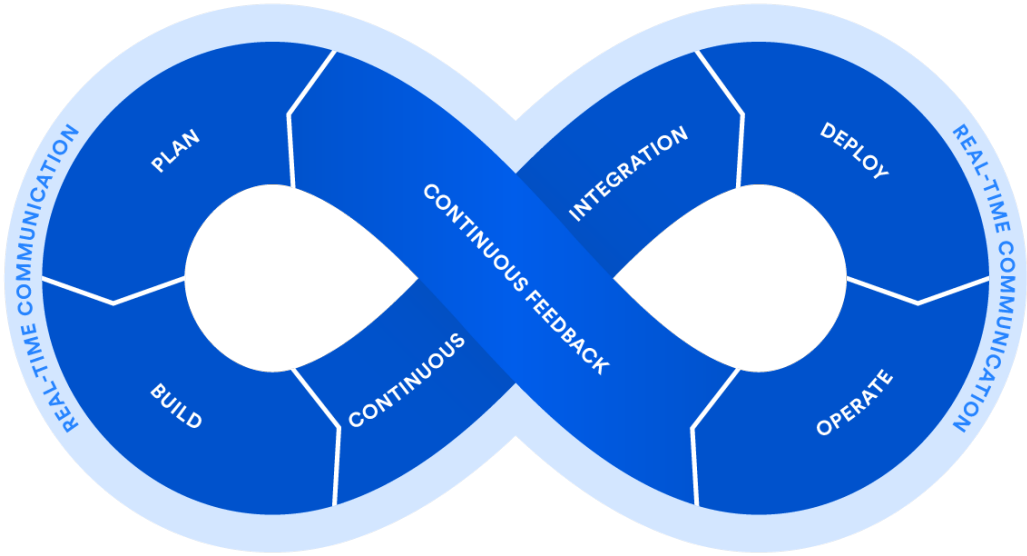


DEVOPS



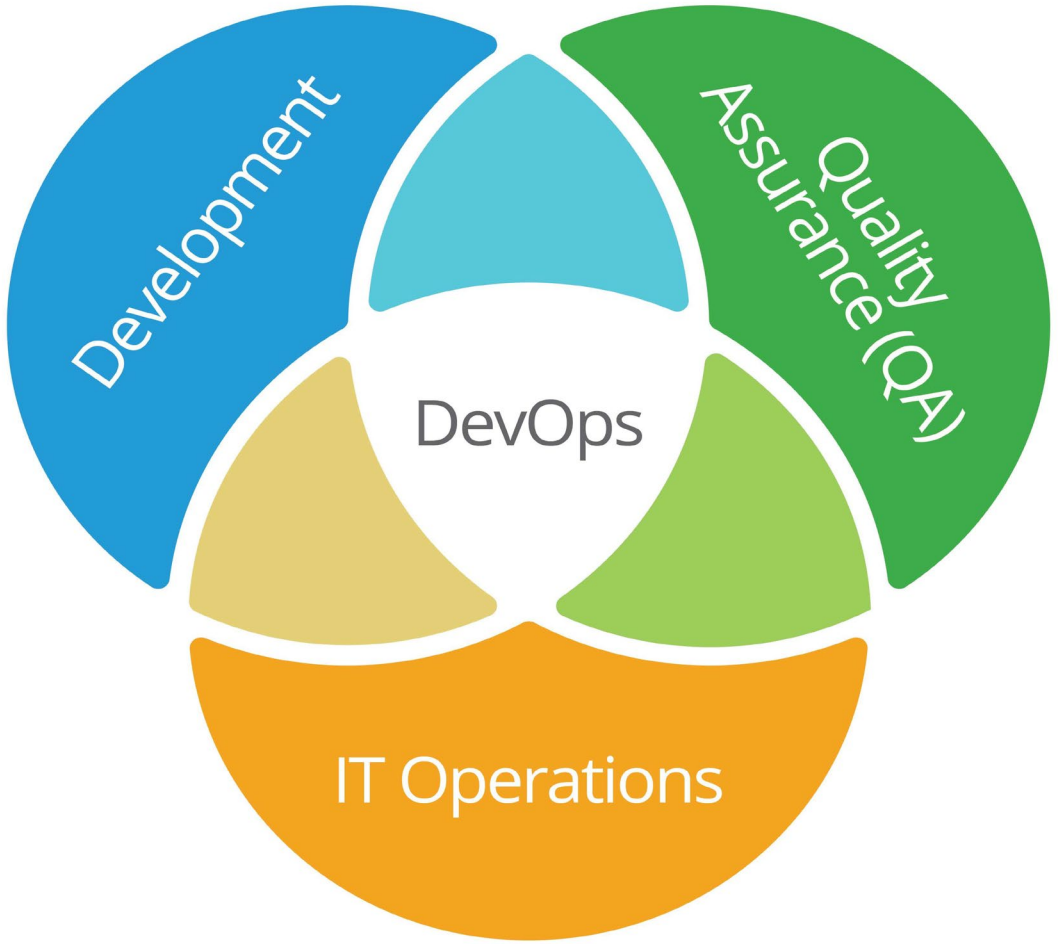
DEVOPS

- Driven by virtualization and Infrastructure as code
- Dev and Ops had been two separate worlds
  - Dev was generally automated
  - Ops was manual and bare metal
- Virtualization turned it all into code
  - Now the same tools can be used in the entire life cycle of a software product
  - Opportunity for full process automation support
- Enables immediate feedback loops between dev and ops



THE GOAL OF DEVOPS

- Desilo-ize the three areas in software development
- Get everyone using the same sorts of tools, practices and automation
  - Better collaboration
  - More control over the entire lifecycle of software



POSITIVE IMPACT ON RISK AND RESILIENCE

- Faster detection and response to risk
  - Continuous integration and monitoring mean issues are identified and remediated early
  - Automated testing and deployment reduce human error during releases
  - Real-time observability (logs, metrics, alerts) improves situational awareness and shortens the mean time to detect (MTTD) and mean time to recover (MTTR)
- Resilience gain
  - Systems and teams become more responsive and adaptive to incidents

POSITIVE IMPACT ON RISK AND RESILIENCE

- Improved change management and reliability
  - Frequent, smaller deployments reduce the risk of large, disruptive failures
  - Automated pipelines ensure consistency and repeatability in builds and releases
  - Version control of IoC provides traceability and rollback capability.
- Risk reduction
  - Fewer surprises in production; controlled, auditable change processes.

POSITIVE IMPACT ON RISK AND RESILIENCE

- Strengthened organizational resilience through culture
  - Promotes cross-functional collaboration, transparency, and shared accountability
  - Integration of dev and ops teams foster collective ownership of uptime, security, and quality.
  - Post-incident reviews (blameless retrospectives) encourage learning and continuous improvement
- Resilience gain
  - The organization learns faster and recovers more intelligently from disruptions

POSITIVE IMPACT ON RISK AND RESILIENCE

- Scalability and elasticity
  - Cloud-native and container-based DevOps infrastructures support scaling up or down quickly during demand spikes or disruptions
  - Disaster recovery automation and multi-region deployments improve system continuity
- Resilience gain
  - Systems can absorb shocks and continue operation under stress

NEGATIVE IMPACT ON RISK AND RESILIENCE

- Increased complexity and interdependence
  - Automation, microservices, and distributed systems add layers of technical complexity
  - Failures in one component (e.g., CI/CD pipeline, cloud dependency) can cascade
- Risk increase
  - More potential failure points; harder to diagnose and contain incidents

NEGATIVE IMPACT ON RISK AND RESILIENCE

- Over-reliance on automation
  - Automation can also amplify human error
    - An incorrect configuration or flawed script can propagate instantly across environments
  - Skill gaps in managing automated pipelines or IaC can increase operational risk
- Risk increase
  - “Automation blindness” where people stop noticing when systems drift off course

NEGATIVE IMPACT ON RISK AND RESILIENCE

- Cultural and governance challenges
  - DevOps culture requires trust, openness, and collaboration which can be difficult shifts for hierarchical organizations
  - Lack of clear accountability can introduce governance and compliance risks
- Risk increase
  - Inconsistent implementation of policies, especially in regulated industries

NEGATIVE IMPACT ON RISK AND RESILIENCE

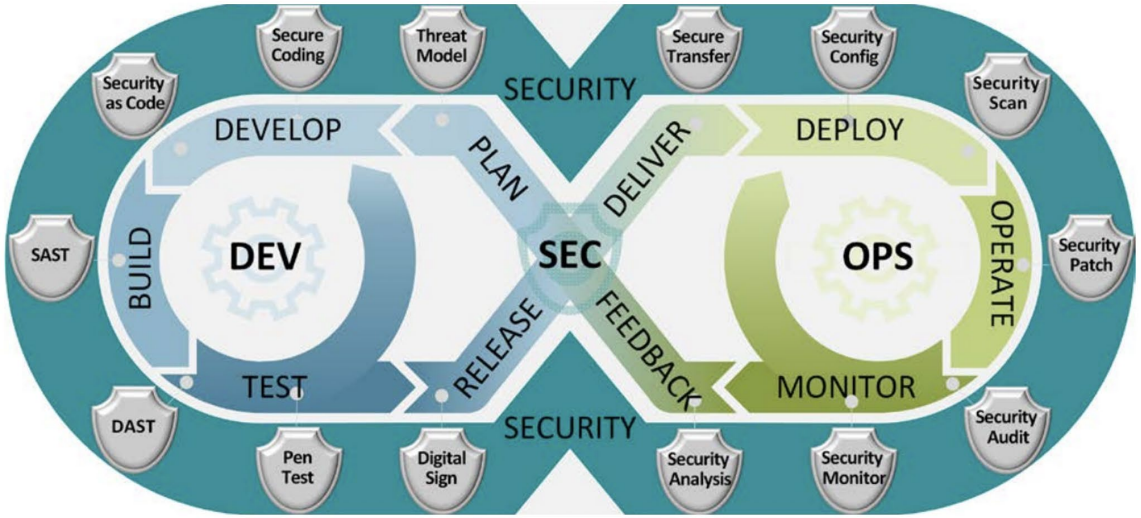
- Tooling and vendor dependency
  - Reliance on cloud platforms and CI/CD tools introduces vendor lock-in and third-party risk
  - A compromise or outage in a key DevOps service can disrupt entire delivery pipelines
- Risk increase
  - Dependency on external resilience, not just internal robustness

NEGATIVE IMPACT ON RISK AND RESILIENCE

- Continuous change fatigue
  - Constant deployment/monitoring cycles can lead to cognitive overload and burnout in teams
  - If not managed with sustainable pace principles, resilience can erode over time
- Risk increase
  - Human resilience, not just technical resilience, may degrade

DEVSECOPS

- DevSecOps (Development, Security, and Operations)
  - Integrates security practices into every phase of the development and delivery pipeline
- Core Idea
  - Traditional DevOps improves speed and collaboration, but often risks overlooking security until late in the process.
  - DevSecOps shifts security left
  - Embedding automated security checks, compliance controls, and risk management early and continuously throughout the lifecycle



DEVSECOPS KEY PRINCIPLES

- Shift security left
  - Incorporate security reviews, code scans, and threat modeling at the earliest stages
  - Prevent vulnerabilities rather than reacting to them after release
- Automation of security controls
  - Security testing becomes part of the CI/CD pipeline
    - For example: static code analysis, dependency scanning, container image checks
  - Automated policy enforcement ensures compliance without slowing down delivery
- Collaboration and shared responsibility
  - Security, Dev, and Ops teams collaborate to build secure systems by design
  - Security champions often embed within development teams to guide best practices

DEVSECOPS KEY PRINCIPLES

- Continuous monitoring and feedback
  - Runtime monitoring, logging, and incident response are integrated to detect and respond to threats quickly
  - Feedback loops turn incidents into learning opportunities
- Security as code / policy as code
  - Infrastructure, configurations, and security policies are version-controlled, tested, and able to be audited
  - Improves traceability and recovery

DEVSECOPS KEY PRINCIPLES

Positive Impacts	Potential Challenges
Early vulnerability detection reduces exposure.	Increased tool and process complexity.
Continuous compliance improves audit readiness.	Requires cultural change — developers must own security.
Automated monitoring enhances resilience to attacks.	Over-automation without understanding can create blind spots.
Embeds resilience by design into the software lifecycle.	Initial setup costs and skills gaps may slow adoption.

CICD BASICS

- CICD is not a methodology
  - It is not Agile or DevOps
  - Although both rely on CICD and use it extensively
- CICD is process automation applied to SE
  - Similar to other kinds of automation
  - Improves process efficiency and effectiveness
- CICD is process agnostic
  - Can be used anywhere a SE process is well defined
  - Using CICD with bad processes makes them worse

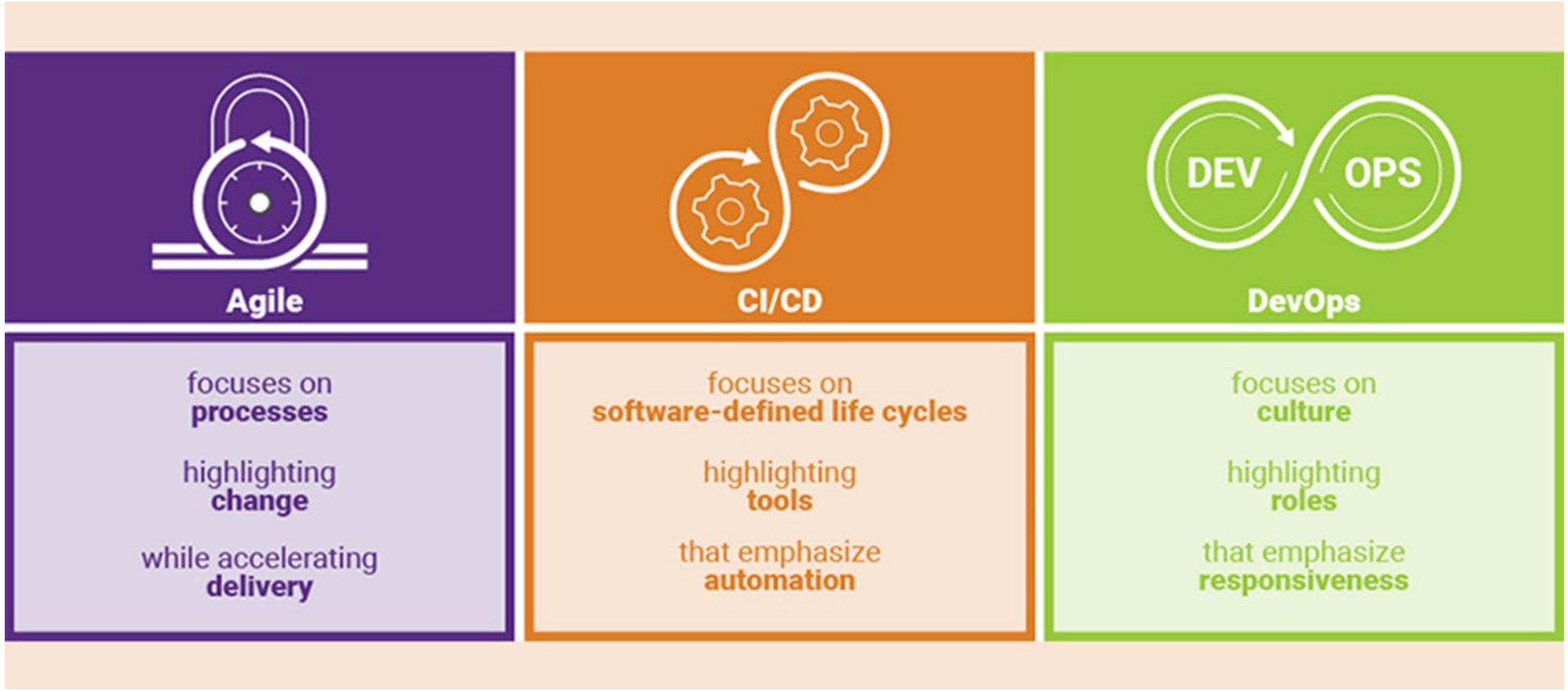
A fool with a tool is still a fool

Martin Fowler

A computer lets you make more mistakes faster than any invention in human history – with the possible exceptions of handguns and tequila

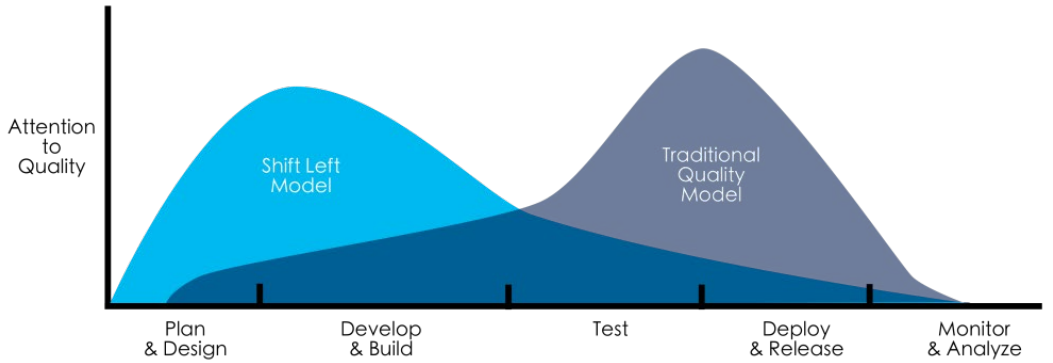
Mitch Ratcliffe

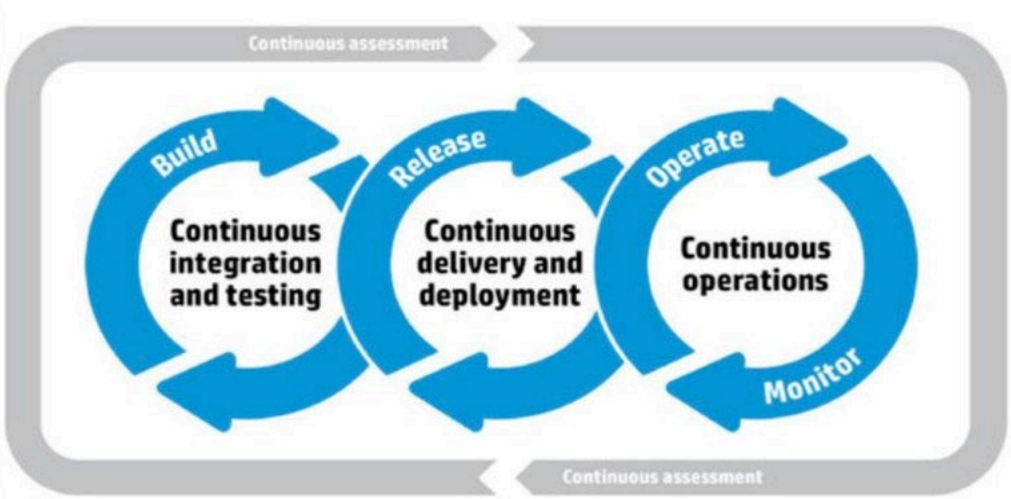
AGILE, DEVOPS AND CICD



CONTINUOUS TESTING

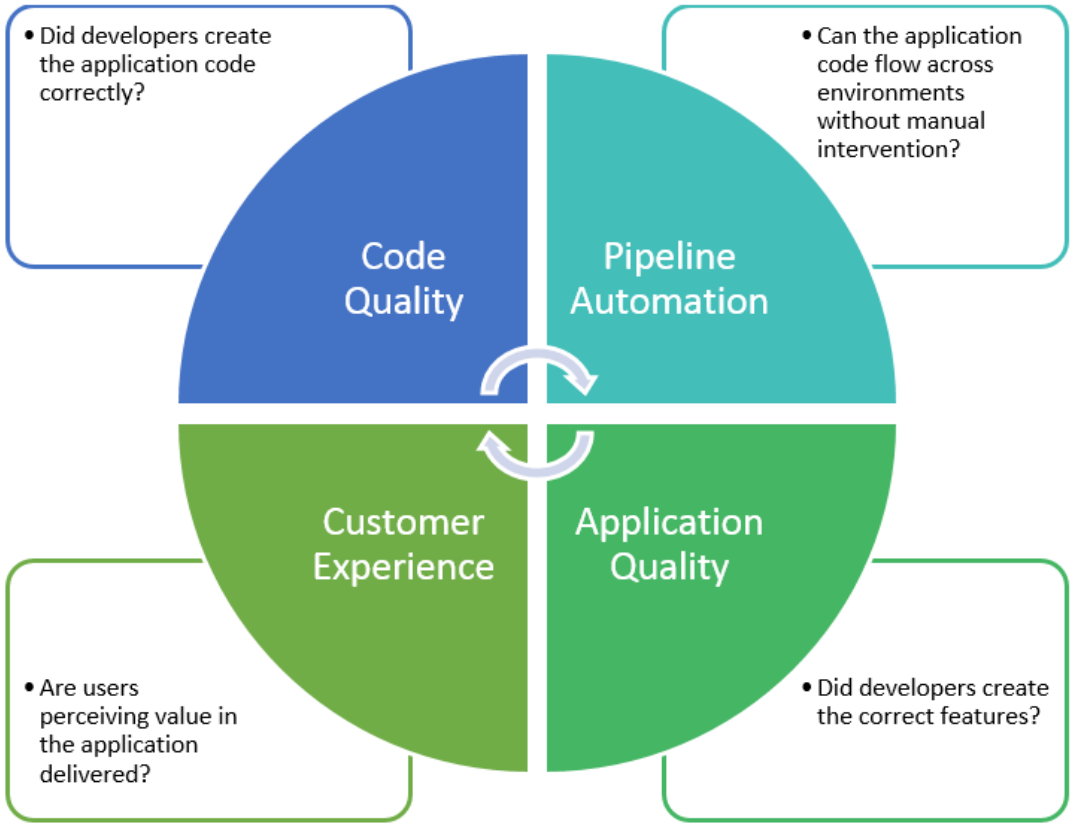
- Continuous Testing
  - Every artifact is tested as it is created
- Shift Left Model
  - Test early, test often
- CICD also adds
  - Automated testing at every stage
- CT is triggered by events in the CICD process
  - Checking in code => automated unit testing
  - Build => integration testing



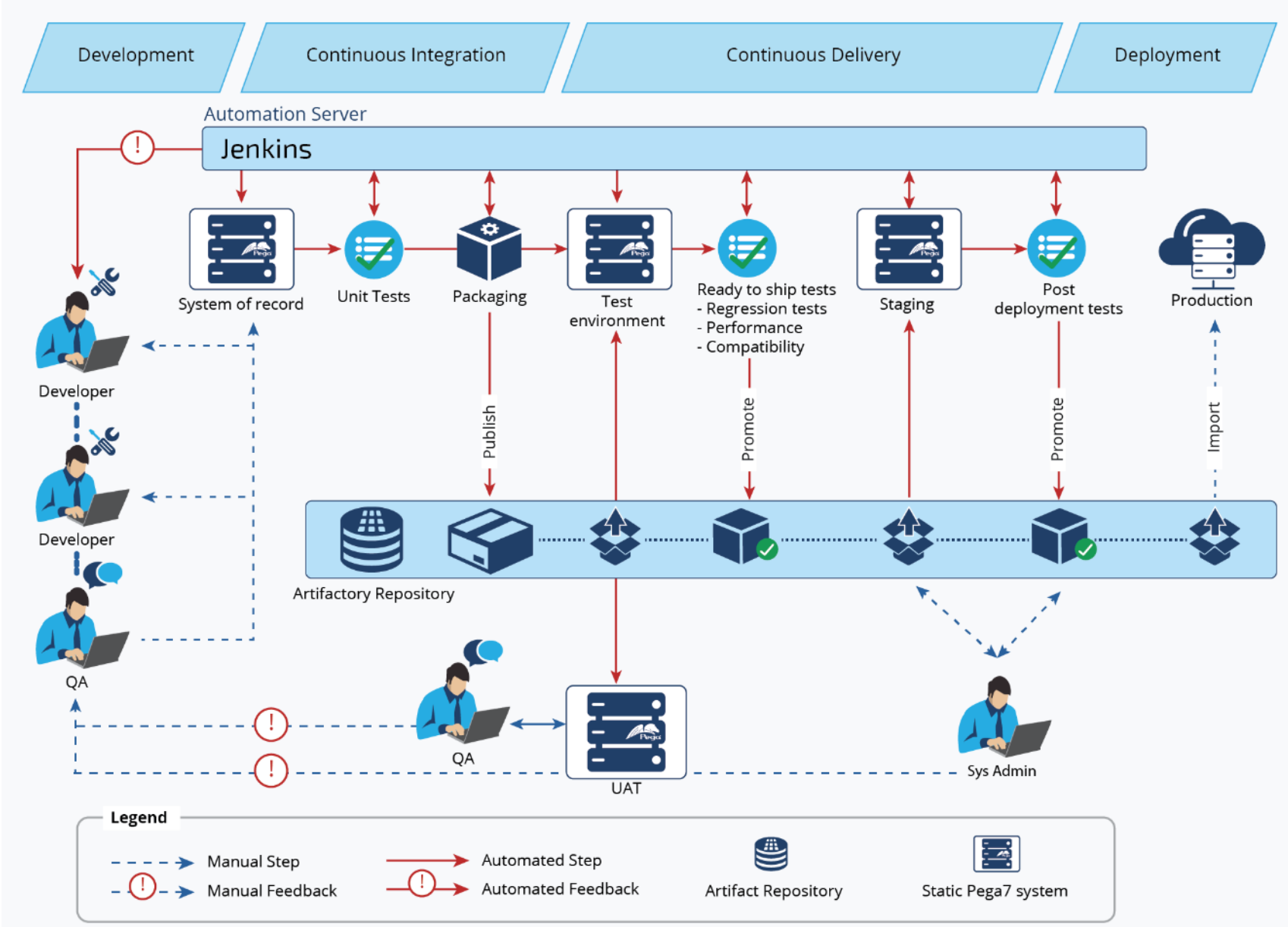


CONTINUOUS TESTING

- Does not replace human based testing
  - Like pair programming and code reviews
- Creates “quality gates”
  - Development pipelines abort when tests fail



PIPELINES



CICD BENEFITS

- Smaller code changes are simpler (more atomic) and have fewer unintended consequences
- Fault isolation is simpler and quicker.
- Mean time to resolution (MTTR) is shorter because of the smaller code changes and quicker fault isolation
- Testability improves due to smaller, specific changes
  - These smaller changes allow more accurate positive and negative tests
- Elapsed time to detect and correct production escapes is shorter with a faster rate of release
- The backlog of non-critical defects is lower because defects are often fixed before other feature pressures arise
- The product improves rapidly through fast feature introduction and fast turn-around on feature changes
- Upgrades introduce smaller units of change and are less disruptive

CICD BENEFITS

- CI-CD product feature velocity is high
  - The high velocity improves the time spent investigating and patching defects
- Feature toggles and blue-green deploys enable seamless, targeted introduction of new production features
- You can introduce critical changes during non-critical (regional) hours
  - This non-critical hour change introduction limits the potential impact of a deployment problem
- Release cycles are shorter with targeted releases and this blocks fewer features that aren’t ready for release
- End-user involvement and feedback during continuous development leads to usability improvements
  - You can add new requirements based on customer’s needs on a daily basis

CICD RISK DOWNSIDES

- Change volume and cognitive load risks
  - Downside of smaller code changes, faster releases, rapid feature introduction
  - Change saturation
    - The number of changes increases, which can overwhelm teams, reviewers, and operations
  - Context loss
    - Engineers may lose the broader system context when working on highly atomic changes, leading to suboptimal design decisions
  - Review fatigue
    - High-frequency pull requests can result in shallow reviews or rubber-stamping
  - Traceability gaps
    - Mapping many small changes to business outcomes, requirements, or incidents becomes harder without strong tooling

CICD RISK DOWNSIDES

- Quality and testing risks
  - Downside of improved testability, better positive/negative tests, faster detection
  - False confidence in automation
    - Heavy reliance on automated tests can mask gaps in test coverage
    - For example: edge cases, usability, performance
  - Test debt accumulation
    - Rapid delivery can cause poorly designed or flaky tests to accumulate
  - Overfitting tests to implementation
    - Small changes may lead to tests that validate how code works, not what the system should do
  - Reduced exploratory testing
    - Speed pressure may reduce time spent on human-driven testing and system-level validation.

CICD RISK DOWNSIDES

- Production stability risks
  - Downside of faster MTTR, quicker fault isolation, smaller upgrades
  - Increased production volatility
    - Frequent releases increase the likelihood of customer-facing changes, even if individual changes are small
  - Failure blast radius underestimation
    - A small change can still have systemic impact in tightly coupled architectures
  - Rollback complexity
    - With many interdependent small changes, rolling back a single issue may not be straightforward
  - Configuration drift
    - CI/CD pipelines that deploy frequently may exacerbate environment inconsistencies

CICD RISK DOWNSIDES

- Security and compliance risks
  - Downside of faster release cycles, smaller units of change
  - Security review bypass risk
    - Speed may lead to reduced manual security reviews or skipped threat modeling
  - Vulnerability propagation
    - Vulnerabilities can spread rapidly to production if not detected early
  - Secrets exposure
    - CI/CD pipelines often require credentials, increasing the risk of secret leakage
  - Audit and compliance gaps
    - High deployment frequency can challenge traditional audit trails, approvals, and segregation-of-duties models

CICD RISK DOWNSIDES

- Incident and operational risks
  - Downside of faster detection and resolution
  - Alert fatigue
    - Frequent changes can generate frequent alerts, reducing signal-to-noise ratio
  - Root cause ambiguity
    - When multiple changes are deployed in close succession, identifying the true root cause can still be difficult
  - Operational dependency risk
    - Operations teams become tightly dependent on CI/CD tooling availability and correctness
  - Pipeline failure impact
    - CI/CD pipeline outages can block all releases, becoming a single point of failure

CICD RISK DOWNSIDES

- Governance and control risks
  - Downside of reduced backlog, rapid improvement
  - Erosion of change control discipline
    - Traditional approval gates may be weakened or removed without adequate replacement controls
  - Inconsistent standards enforcement
    - Without strong policy-as-code, teams may diverge in quality, security, or documentation practices
  - Shadow releases
    - Teams may deploy changes without full stakeholder awareness due to automation opacity
  - Metrics misalignment
    - Overemphasis on deployment frequency can overshadow stability, customer impact, or long-term quality

CICD RISK DOWNSIDES

- Organizational and cultural risks
  - Downside of fast turnaround, rapid product improvement
  - Burnout risk
    - Constant delivery pressure can increase stress and reduce sustainability
  - Skill gap exposure
    - CI/CD requires strong engineering discipline; immature teams may struggle
  - Blame amplification
    - High visibility of frequent changes can lead to unhealthy blame cultures if incidents occur
  - Uneven ownership
    - Faster cycles can blur accountability between development, QA, and operations

CICD RISK DOWNSIDES

- Architecture and technical debt Risks
  - Downside of smaller upgrades, faster feature changes
  - Local optimization risk
    - Small changes may optimize locally while degrading overall system architecture
  - Deferred refactoring
    - Teams may repeatedly ship incremental fixes instead of addressing structural issues
  - Coupling amplification
    - CI/CD exposes tight coupling faster but does not fix it; frequent breakages may occur
  - Backward compatibility erosion
    - Rapid iteration can unintentionally break consumers or integrations

Q&A AND OPEN DISCUSSION