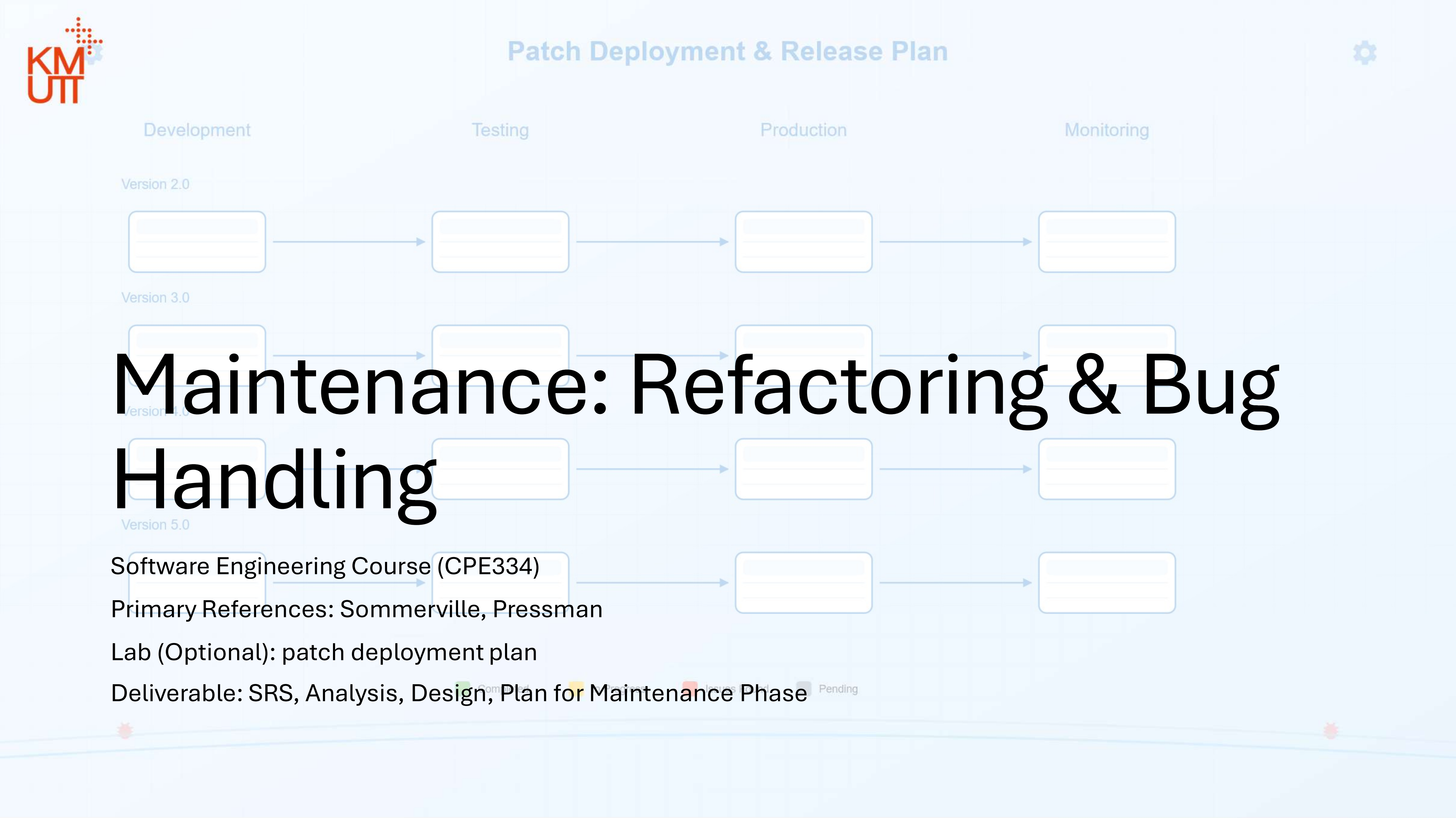


Maintenance: Refactoring & Bug Handling

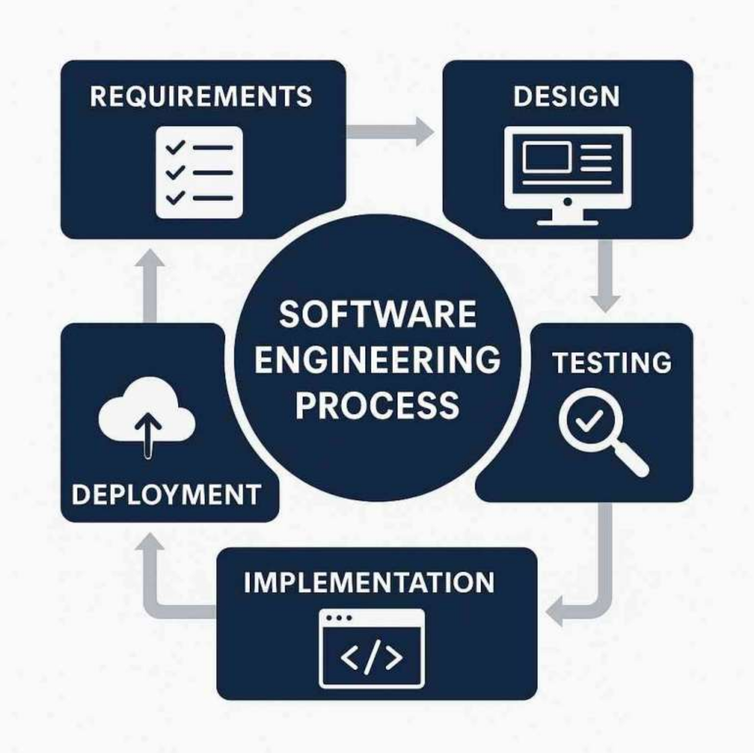
Software Engineering Course (CPE334)

Primary References: Sommerville, Pressman

Lab (Optional): patch deployment plan

Deliverable: SRS, Analysis, Design, Plan for Maintenance Phase

Recap week 01-05, 07-10



🏗️ Phase 1: Overview & Requirements (Week 1, 2, 3)

Week	Lecture Focus	Key Takeaways
1	Intro to Software Engineering & SDLC Models	**SE**: Principles for creating quality software. **SDLC**: Software Development Lifecycle (Plan, Design, Implement, Test, Deploy, Maintain). **Models**: Understanding Waterfall, Agile, and Spiral methodologies.
2	Requirements Engineering: Elicitation & Specification	**Elicitation**: The process of gathering requirements from Stakeholders (Interviews, Prototyping, Observation). **Analysis**: Checking requirements for feasibility and consistency.
3	Requirements Modeling & Prioritization	**Modeling**: Creating visual representations of requirements (e.g., Use Cases, Data Flow Diagrams) to clarify understanding. **Prioritization**: Techniques (MoSCoW, Cost-Value) to determine which requirements are implemented first based on value and risk. **Output**: The prioritized requirements feed into the SRS (Software Requirement Specification).

📐 Phase 2: Architecture & Design (Week 4, 5)

Week	Lecture Focus	Key Takeaways
4	Software Architecture: Patterns & High-Level Design	**Architecture**: The overall structure of the system. **Patterns**: Key architectural styles like Layered (dividing into Presentation, Logic, Data layers) and Microservices (small, independent services).
5	Design Principles & UML (Class, Sequence, State Diagrams)	**Design Principles**: Good design practices. **UML Diagrams**: Class Diagram (static structure), Sequence Diagram (time-ordered interactions), and State Diagram (object status and transitions).

🛠️ Phase 3: Implementation & Control (Week 7, 8)

Week	Lecture Focus	Key Takeaways
7	Project Planning: Estimation & Scheduling	**Planning & Scheduling**: Defining scope and converting the plan into an operating timeline. **Estimation Techniques**: Methods to forecast effort/cost, including Expert Judgment, Three-Point Estimation, and Analogous Estimation. **Project Structure**: Using a Work Breakdown Structure (WBS) to organize tasks.
8	Process Implementation: Version Control & Dev Workflow	**VCS (Git)**: System used to track and manage code changes. **Concepts**: Repository, Branching, Merging for efficient collaboration and the ability to rollback versions. **Workflow**: Utilizing branching strategies (e.g., Gitflow) to coordinate team workstreams.

🔬 Phase 4: Quality & Testing (Week 9, 10)

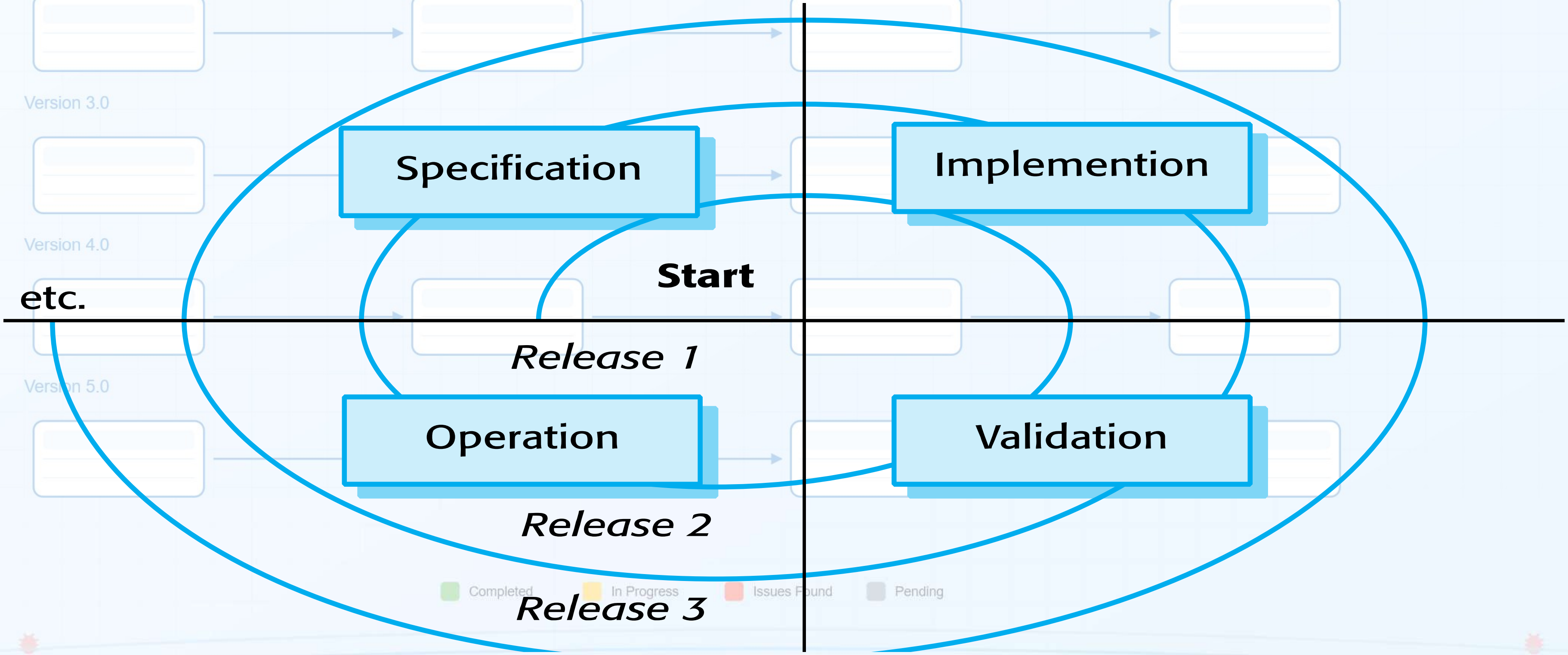
Week	Lecture Focus	Key Takeaways
9	Implementation Practices: Code Quality & Documentation	**Code Quality**: Code that is maintainable, reliable, and meets standards. **Standards**: Coding best practices enforced through Code Reviews and Static Analysis Tools. **Documentation**: Creating essential code documentation (Internal/API Docs).
10	Software Testing: Levels & Techniques	**4 Levels of Testing**: 1. Unit Testing, 2. Integration Testing, 3. System Testing, 4. Acceptance Testing (UAT). **Techniques**: White Box (tests internal structure) vs. Black Box (tests external behavior).

♻️ Phase 5: Maintenance (Week 11)

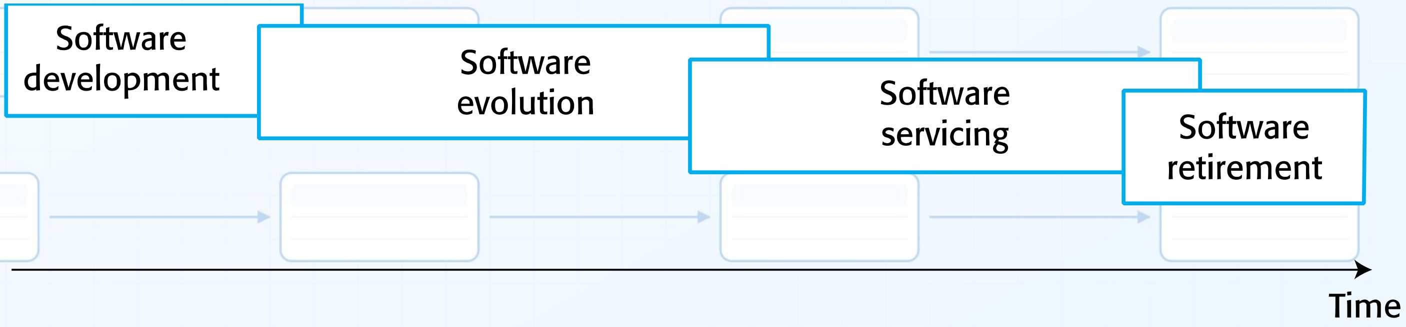
Week	Lecture Focus	Key Takeaways
11 (This week)	Maintenance: Refactoring & Bug Handling	**Software Maintenance**: Post-deployment activities, including Corrective (fixing bugs), Adaptive (adjusting to environment changes), and Perfective (improving functionality/performance). **Refactoring**: Disciplined process of restructuring code to improve quality without changing external behavior. **Bug Handling**: Understanding the Bug Life Cycle and using defect management tools. **Code Smells**: Identifying structural patterns in code that indicate deeper design problems needing Refactoring.

Software Evolution

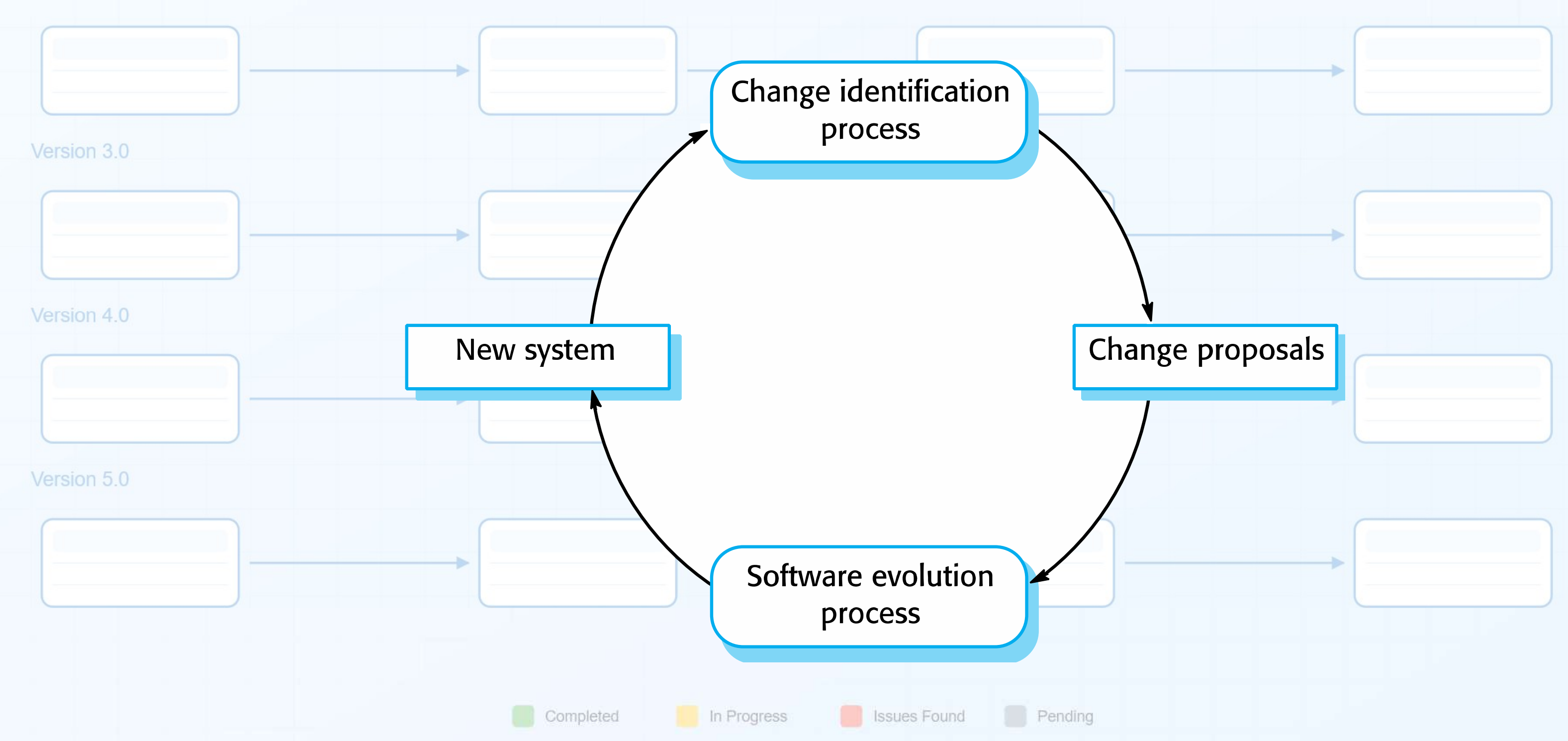
A spiral model of development and evolution



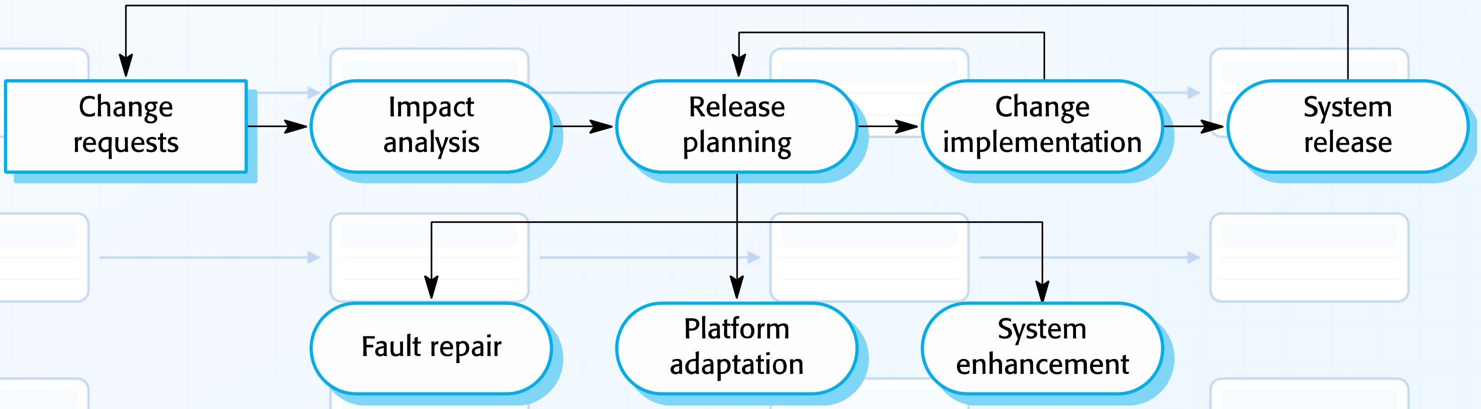
Evolution and servicing



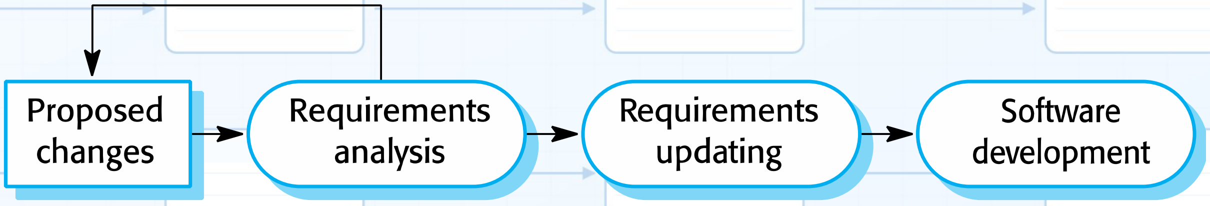
Change identification and evolution processes



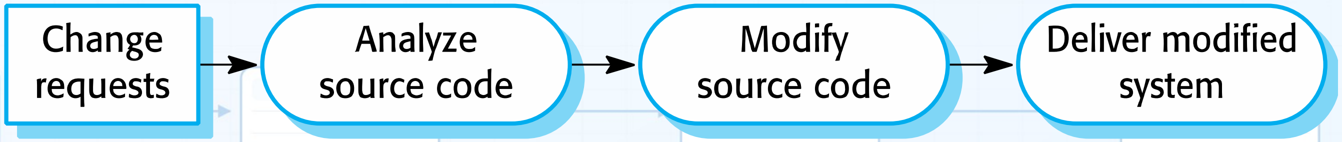
The software evolution process

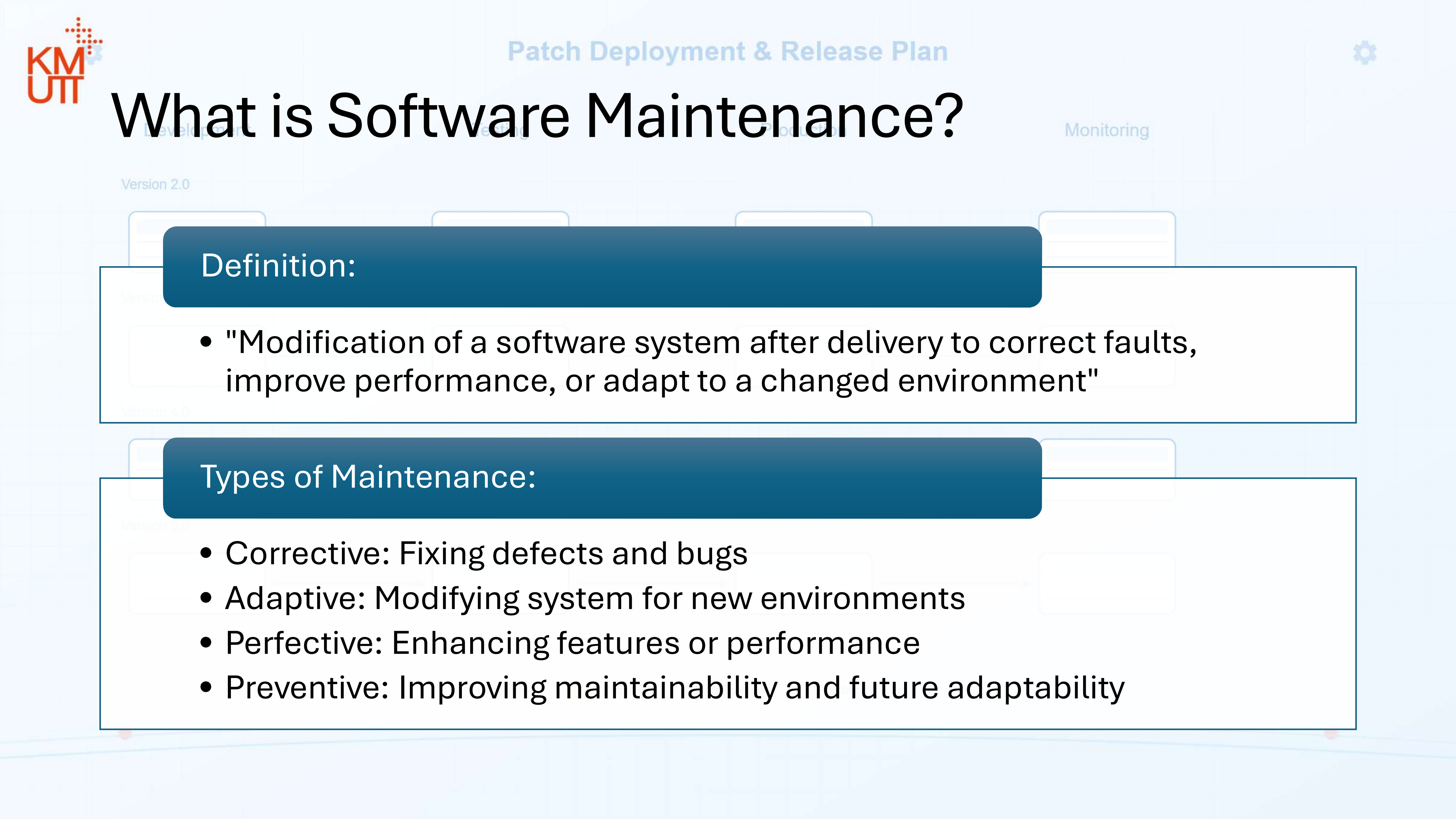


Change implementation



The emergency repair process





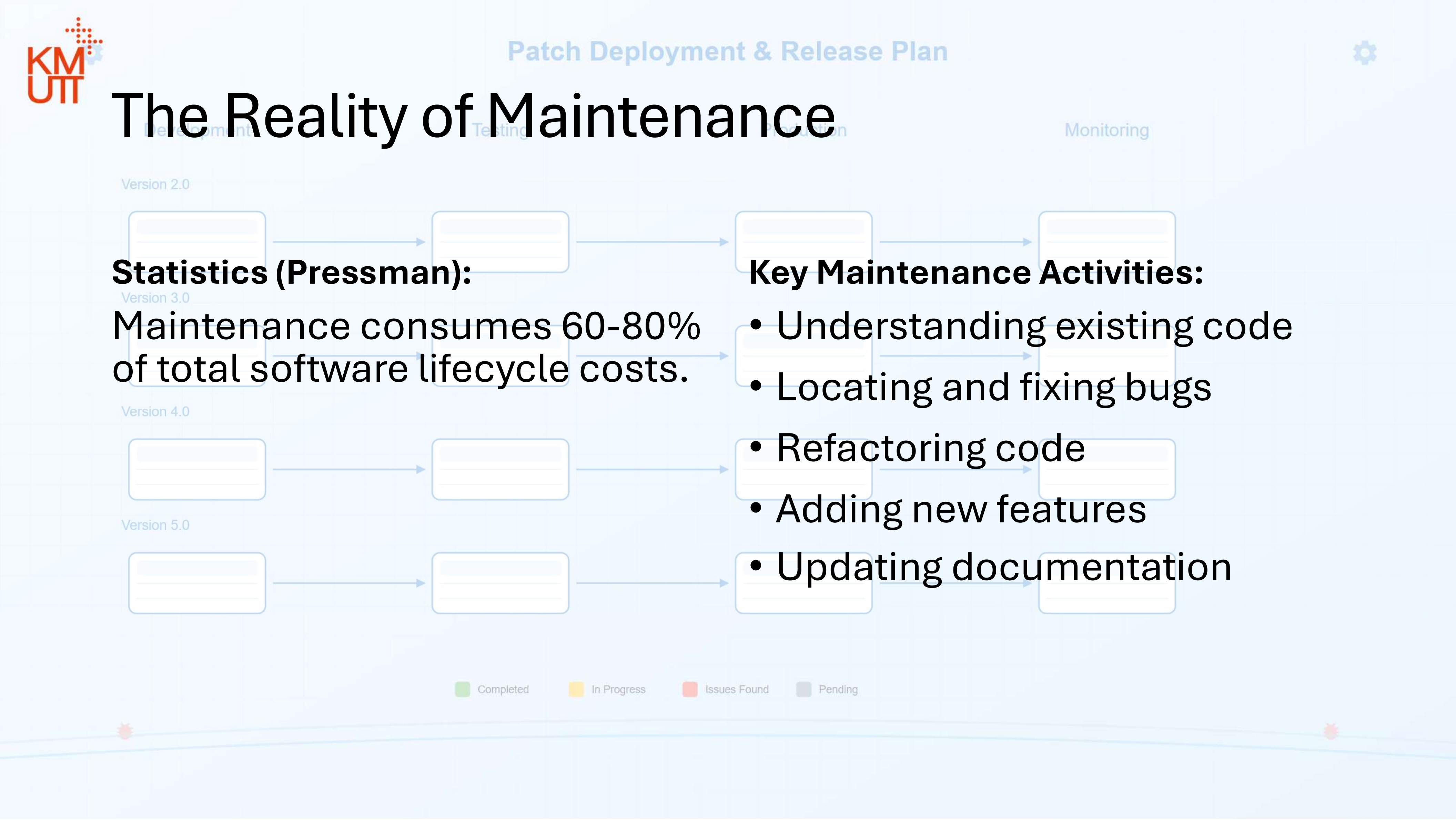
What is Software Maintenance?

Definition:

- "Modification of a software system after delivery to correct faults, improve performance, or adapt to a changed environment"

Types of Maintenance:

- Corrective: Fixing defects and bugs
- Adaptive: Modifying system for new environments
- Perfective: Enhancing features or performance
- Preventive: Improving maintainability and future adaptability



The Reality of Maintenance

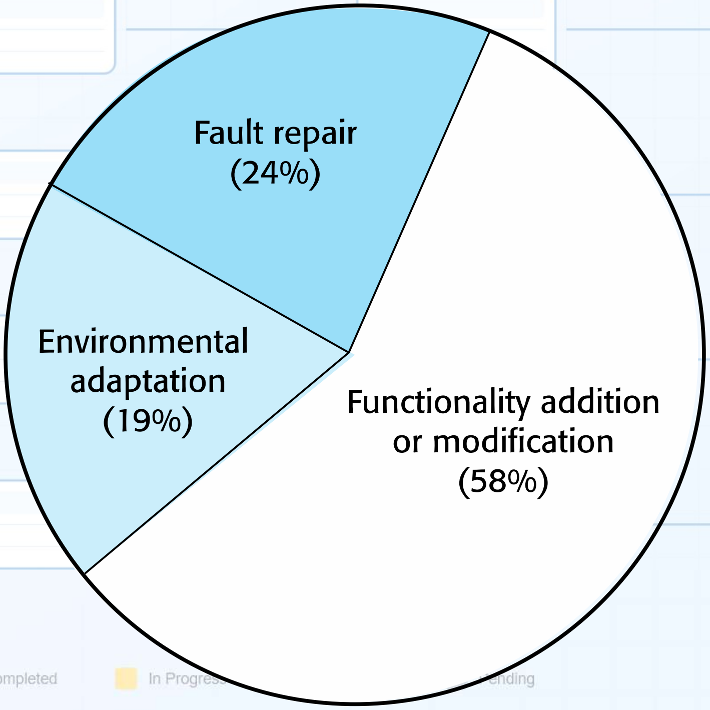
Statistics (Pressman):

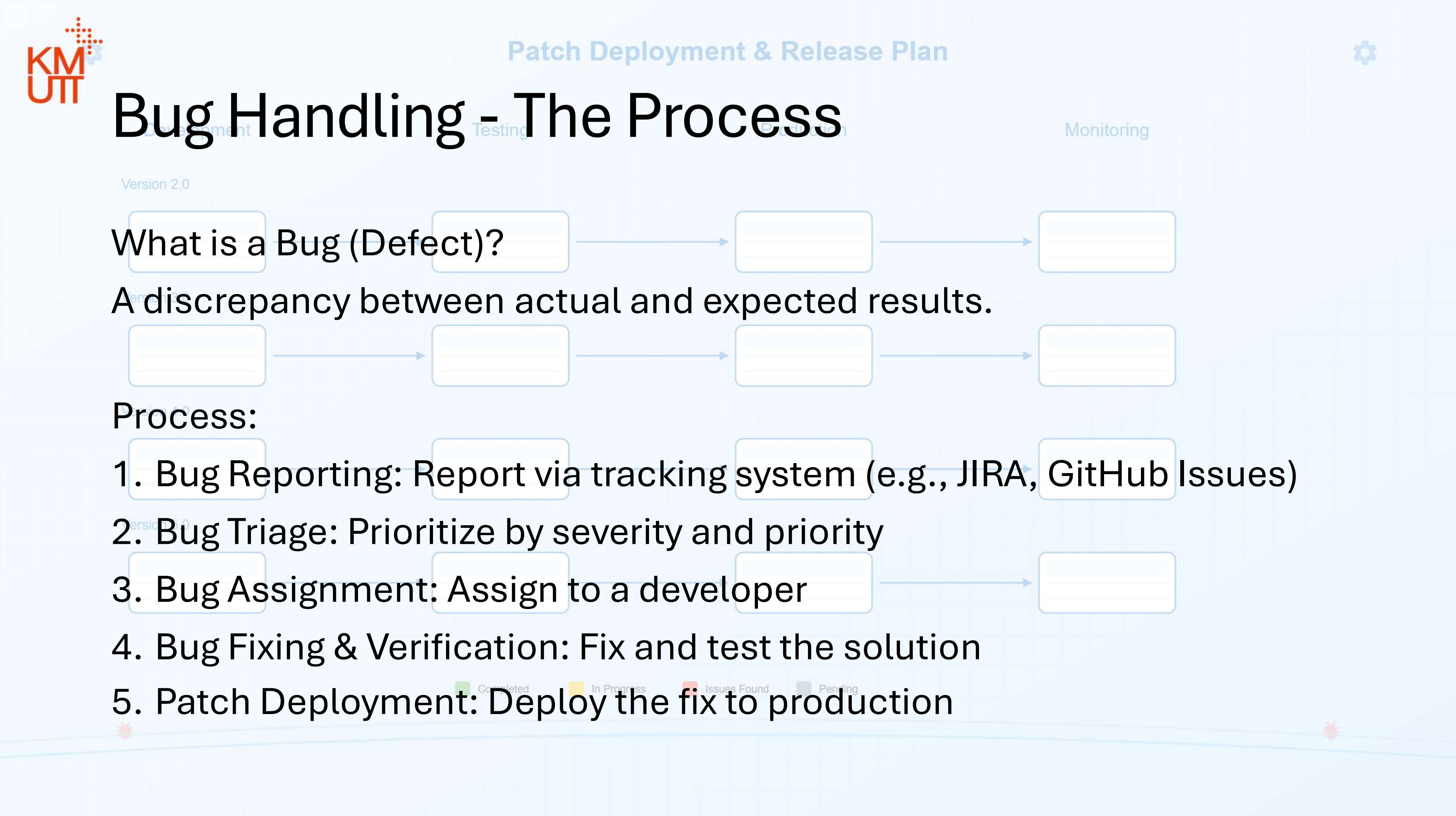
Maintenance consumes 60-80% of total software lifecycle costs.

Key Maintenance Activities:

- Understanding existing code
- Locating and fixing bugs
- Refactoring code
- Adding new features
- Updating documentation

Maintenance effort distribution





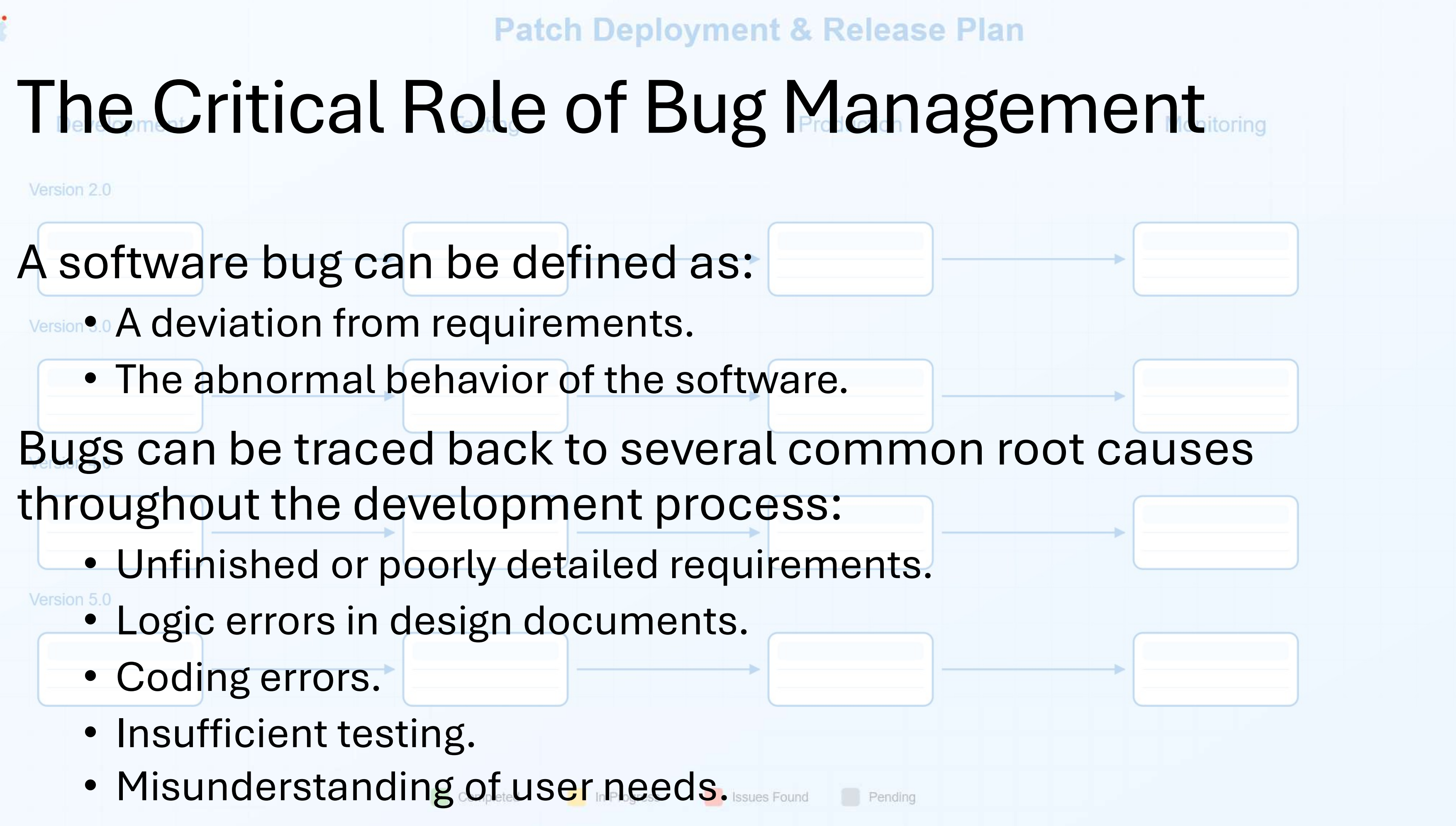
Bug Handling - The Process

What is a Bug (Defect)?

A discrepancy between actual and expected results.

Process:

1. Bug Reporting: Report via tracking system (e.g., JIRA, GitHub Issues)
2. Bug Triage: Prioritize by severity and priority
3. Bug Assignment: Assign to a developer
4. Bug Fixing & Verification: Fix and test the solution
5. Patch Deployment: Deploy the fix to production



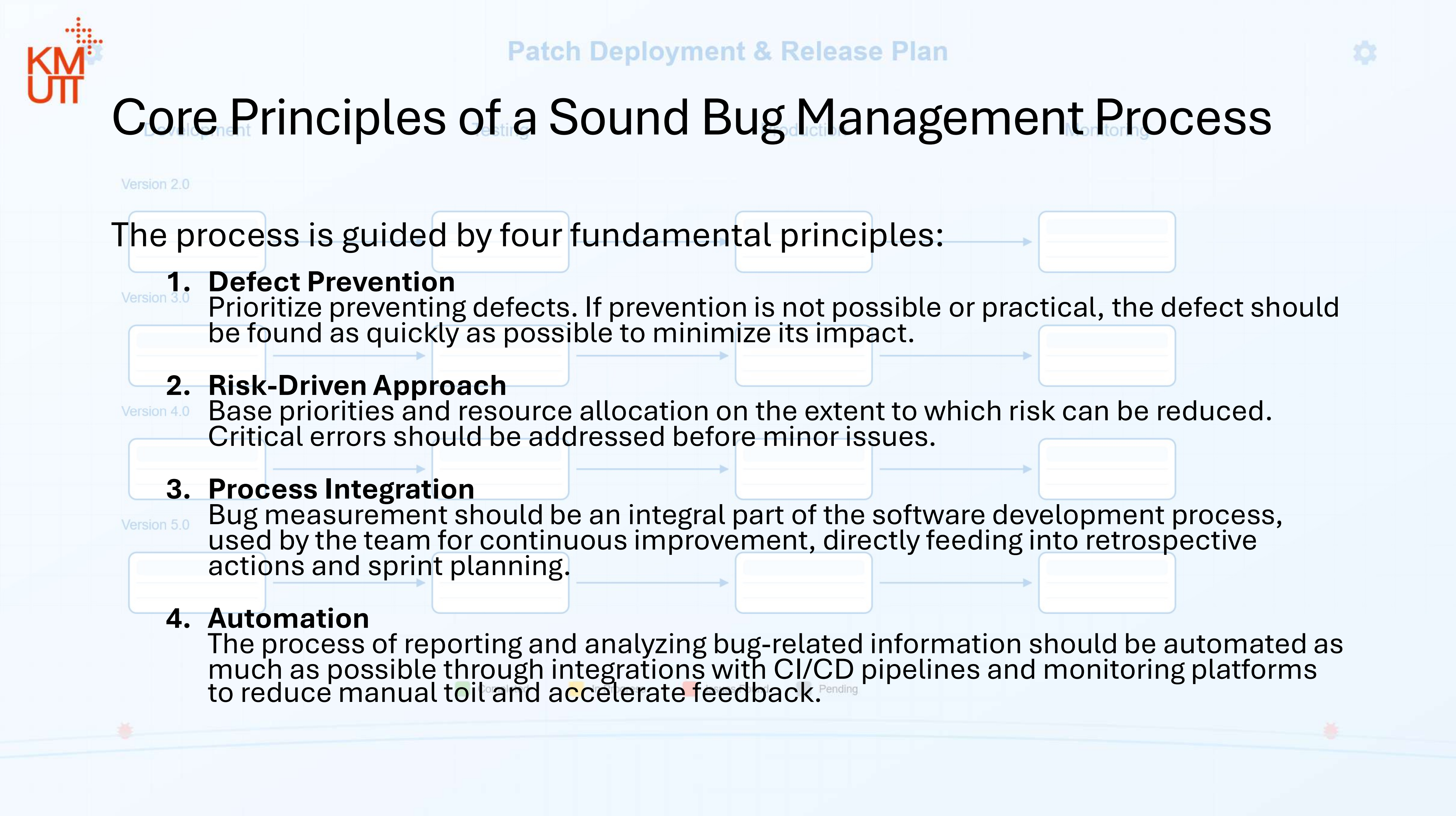
The Critical Role of Bug Management

A software bug can be defined as:

- A deviation from requirements.
- The abnormal behavior of the software.

Bugs can be traced back to several common root causes throughout the development process:

- Unfinished or poorly detailed requirements.
- Logic errors in design documents.
- Coding errors.
- Insufficient testing.
- Misunderstanding of user needs.



Core Principles of a Sound Bug Management Process

The process is guided by four fundamental principles:

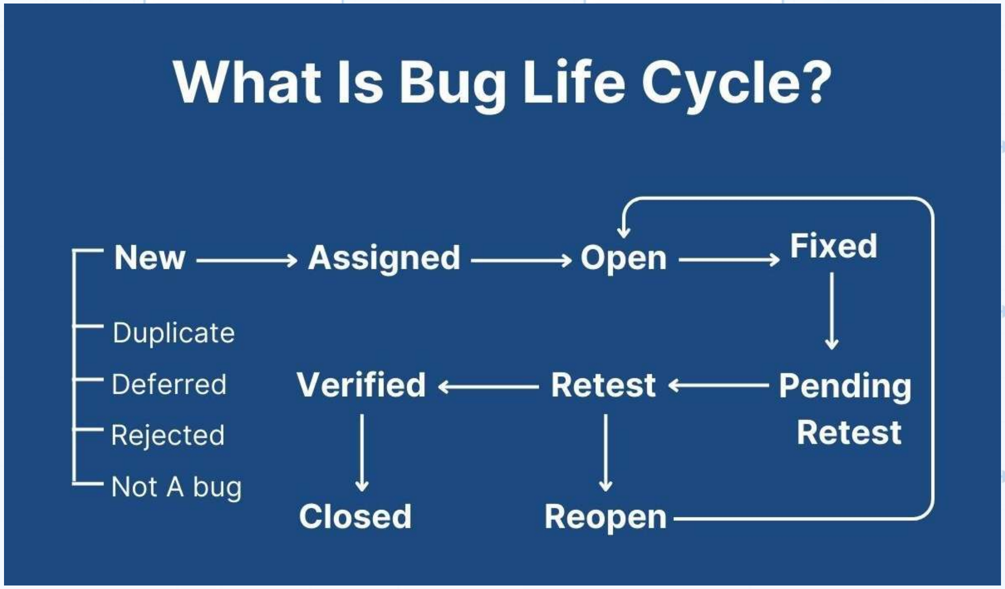
1. **Defect Prevention**
   Prioritize preventing defects. If prevention is not possible or practical, the defect should be found as quickly as possible to minimize its impact.

2. **Risk-Driven Approach**
   Base priorities and resource allocation on the extent to which risk can be reduced. Critical errors should be addressed before minor issues.

3. **Process Integration**
   Bug measurement should be an integral part of the software development process, used by the team for continuous improvement, directly feeding into retrospective actions and sprint planning.

4. **Automation**
   The process of reporting and analyzing bug-related information should be automated as much as possible through integrations with CI/CD pipelines and monitoring platforms to reduce manual toil and accelerate feedback.

The Bug Life Cycle: From Discovery to Resolution



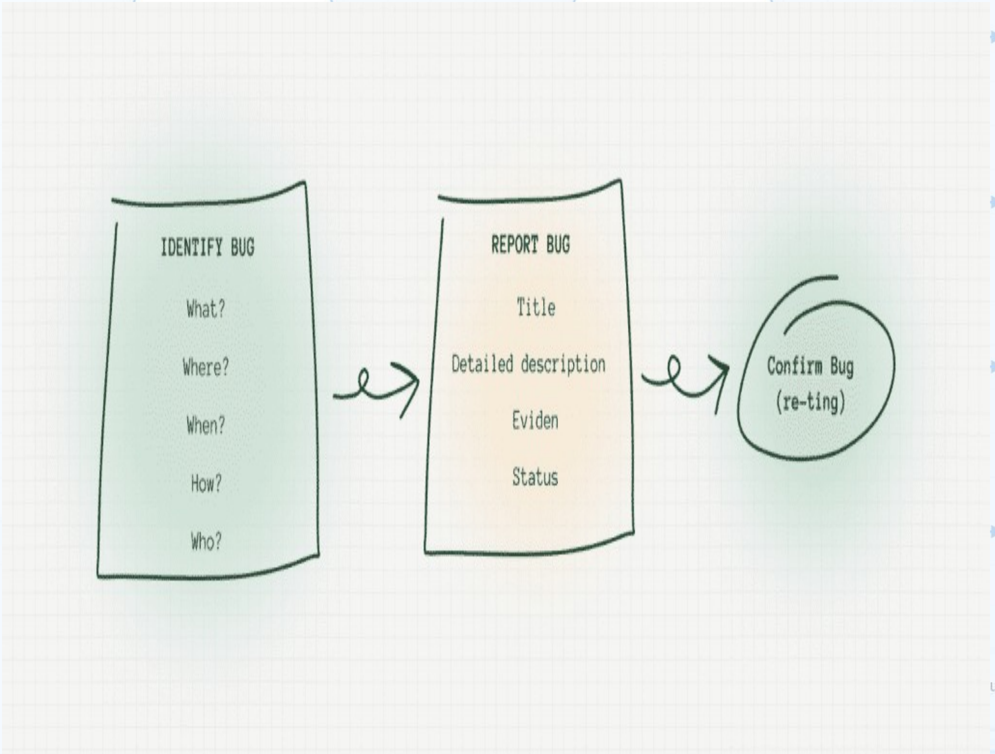
The typical flow of a bug through its life cycle is as follows:

1. **Unconfirmed**: The bug is added, and it’s not yet validated.
2. **New**: The bug is validated, and it must be processed.
3. **Assigned**: The bug is not yet resolved but is assigned to the developer.
4. **Resolved**: The bug is resolved by the developer and is ready for verification.
5. **Verified**: The bug is duly fixed.
6. **Closed**: The bug is fixed and confirmed its absence.

Key Roles & Responsibilities

Role	Description and Responsibilities
Reporter	The person or group who finds and reports the bug.
Test Manager	Person who acknowledges the bug, sets its initial priority and severity, and assigns it to a developer.
Tester	The person or group who tests the bug, tries to reproduce it, and verifies the final solution.
Developer	The person or group who is responsible for resolving the bug.

Anatomy of a Good Bug Report



Essential Components:

- Title/Bug ID: Concise problem summary
- Reporter: Who reported it
- Environment: OS, Browser, Device, Version
- Steps to Reproduce: Detailed, step-by-step instructions
- Expected Result: What should happen
- Actual Result: What actually happens
- Visual Proof: Screenshots, videos, or logs that clearly show the bug in action
- Severity/Priority: Impact and urgency levels
- Evidence: Screenshots, logs, error messages

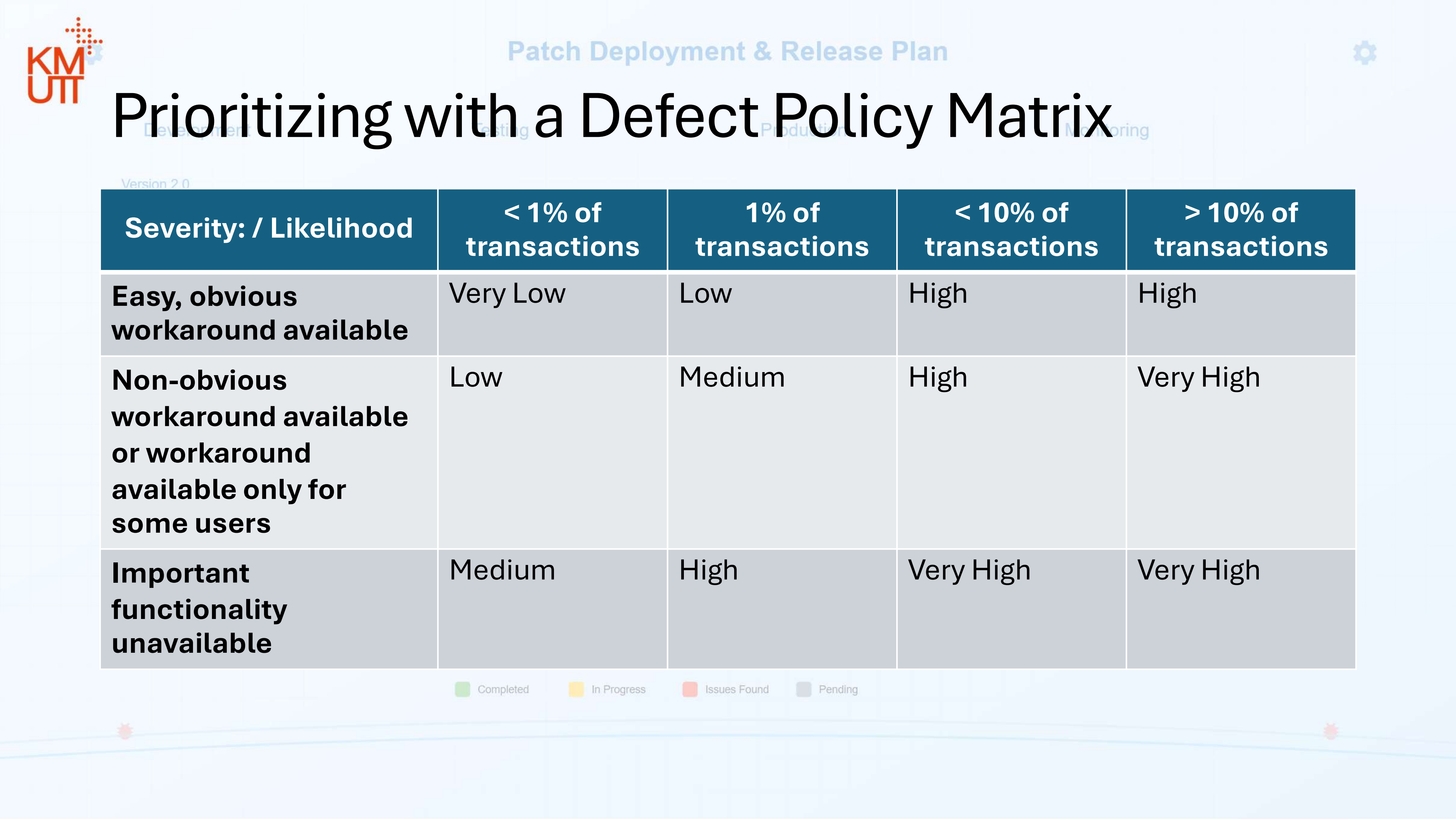
Classifying Bugs: Severity vs. Priority

Bug Severity Levels:

- **Blocker**: Prevents development or testing on the affected product.
- **Critical**: Causes the product’s crash or a function does not work at all.
- **Major**: Affects the product’s feature to be operational.
- **Normal**: Impacts the product to work improperly.
- **Minor**: The product does not work optimally, but a workaround is possible.
- **Trivial**: Irritates the user but does not affect usability.
- **Enhancement**: A proposal for new functionality.

Bug Priority Levels (5=Highest):

- **Immediate (5)**: Blocks work or is a security issue; fix ASAP.
- **Urgent (4)**: Blocks usability of a large portion of the product.
- **High (3)**: Seriously broken but with less impact than Urgent.
- **Normal (2)**: A workaround exists or the functionality is not critical.
- **Low (1)**: The bug is not very important.



Prioritizing with a Defect Policy Matrix

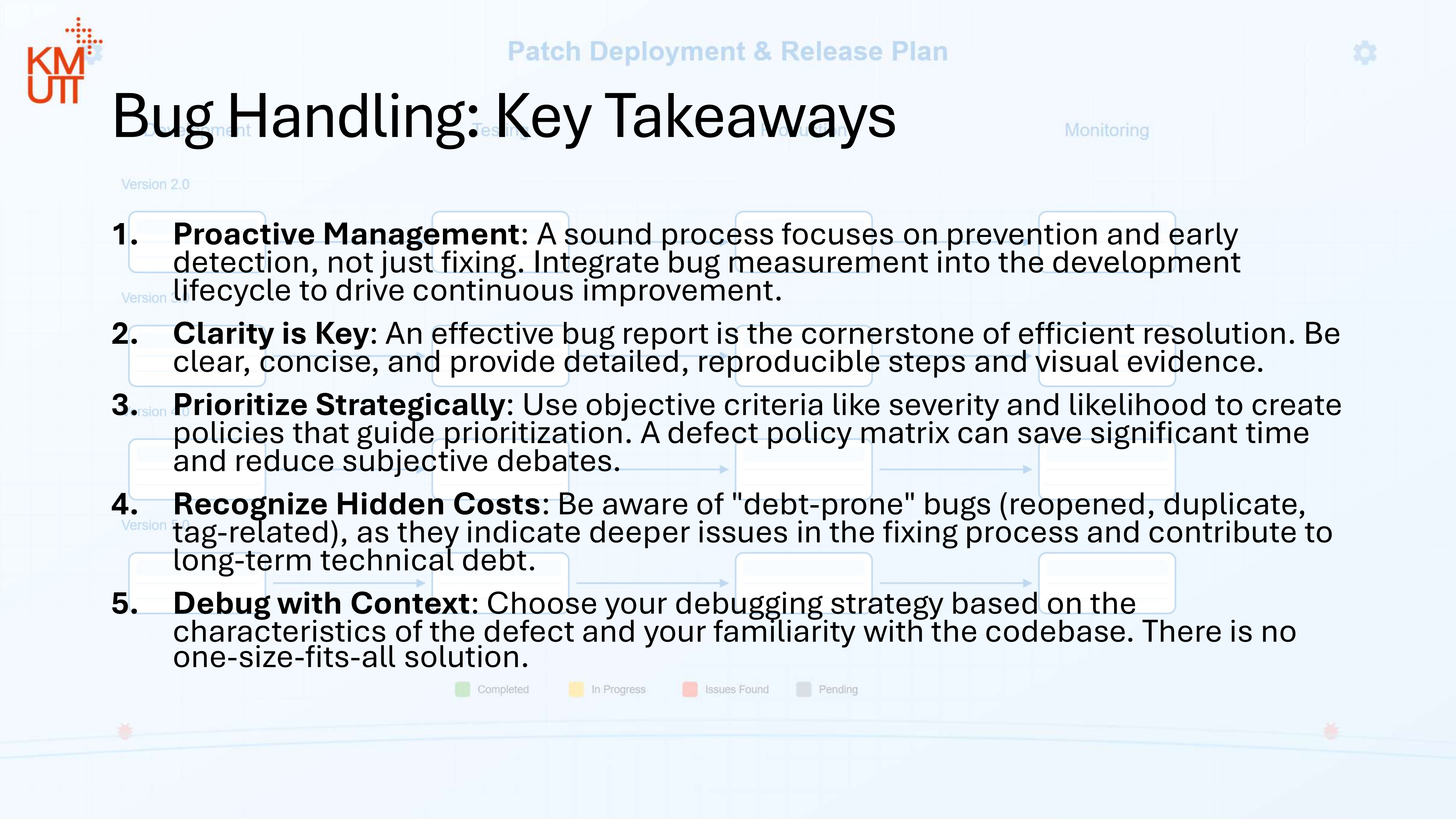
Severity: / Likelihood	< 1% of transactions	1% of transactions	< 10% of transactions	> 10% of transactions
Easy, obvious workaround available	Very Low	Low	High	High
Non-obvious workaround available or workaround available only for some users	Low	Medium	High	Very High
Important functionality unavailable	Medium	High	Very High	Very High

Choosing Your Debugging Strategy

- Primary debugging strategies include:
  - **Hypothesis-test**: Formulate theories about the potential cause and test them by gathering evidence, such as inspecting runtime behavior, logs, or specific code paths.
  - **Backward-reasoning**: Trace the error from its visible symptom backward through the code execution path to uncover the underlying root cause.
  - **Simplification**: Break down the problem into smaller, more manageable parts or remove unnecessary details to isolate the core defect.
  - **Error-message**: Analyze error messages, system logs, and official documentation to understand the content and context of a failure.
  - **Binary-search**: Repeatedly divide the codebase or input space into smaller sections and test each one to systematically isolate the problematic area.
  - **Historical-analysis**: Use version control system tools (like git-bisect) to find the exact commit that introduced a bug by testing versions of the code history.

Bug Handling: Key Takeaways

1. **Proactive Management**: A sound process focuses on prevention and early detection, not just fixing. Integrate bug measurement into the development lifecycle to drive continuous improvement.
2. **Clarity is Key**: An effective bug report is the cornerstone of efficient resolution. Be clear, concise, and provide detailed, reproducible steps and visual evidence.
3. **Prioritize Strategically**: Use objective criteria like severity and likelihood to create policies that guide prioritization. A defect policy matrix can save significant time and reduce subjective debates.
4. **Recognize Hidden Costs**: Be aware of "debt-prone" bugs (reopened, duplicate, tag-related), as they indicate deeper issues in the fixing process and contribute to long-term technical debt.
5. **Debug with Context**: Choose your debugging strategy based on the characteristics of the defect and your familiarity with the codebase. There is no one-size-fits-all solution.



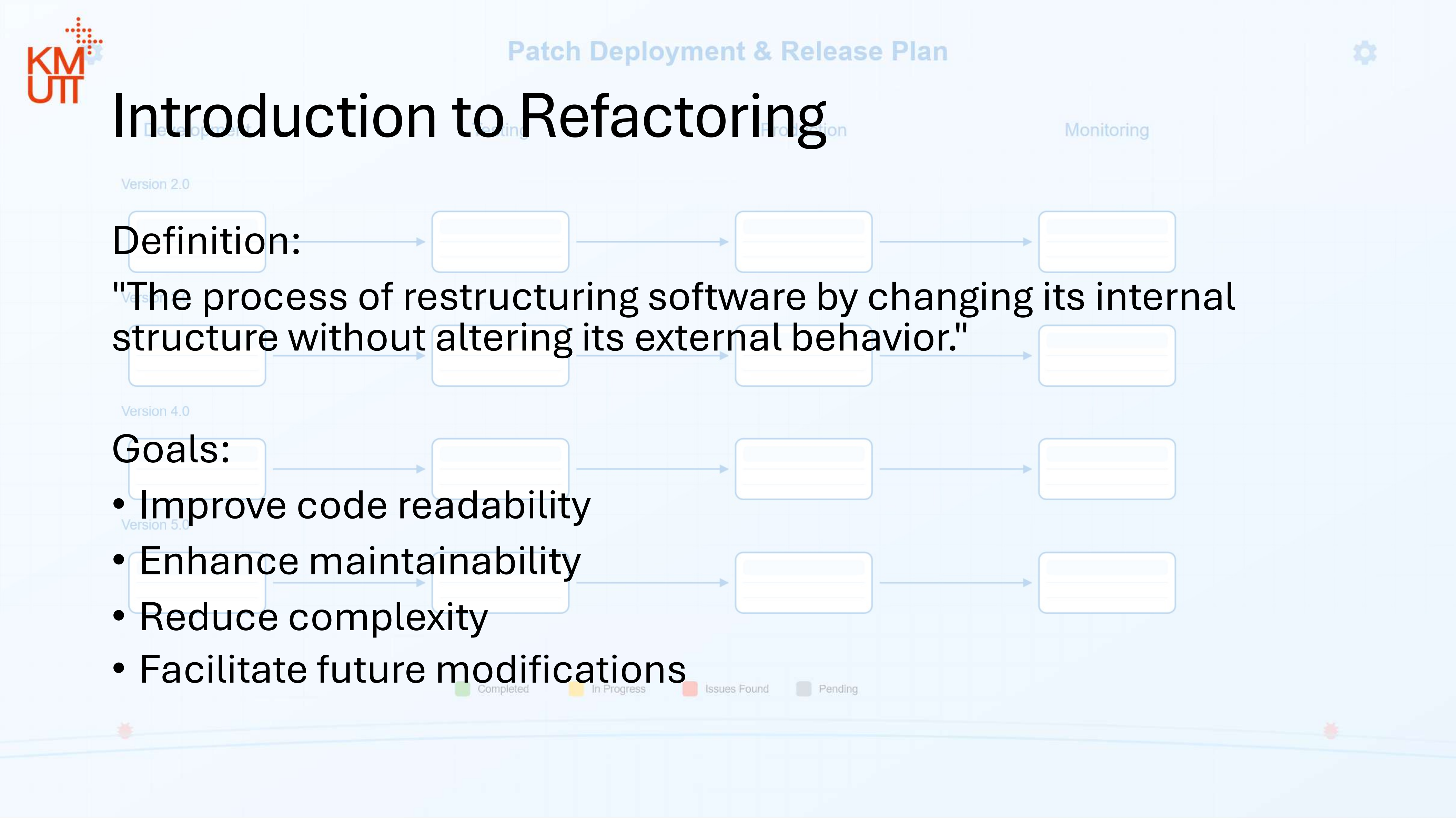
Introduction to Refactoring

Definition:

"The process of restructuring software by changing its internal structure without altering its external behavior."

Goals:

- Improve code readability
- Enhance maintainability
- Reduce complexity
- Facilitate future modifications



Code Smells

What? How can code "smell"??

Well it doesn't have a nose
… but it definitely can stink!

Types Code Smells: Bloaters (Code That Has "Puffed Up" Too Much)

Code or classes that are excessively large or long, making them difficult to manage and understand.

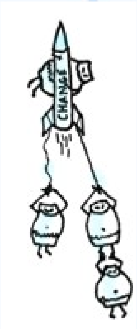
Code Smell	Description	Refactoring Solution
Long Method	A method (or function) with too many lines of code.	🔪 Extract Method (Split into smaller methods).
Large Class	A class with too many fields or methods, covering too many responsibilities.	✂️ Extract Class or Extract Subclass (Separate responsibilities).
Primitive Obsession	Using primitive data types (e.g., int, string) instead of creating a class for that concept.	🎁 Replace Data Value with Object (Create Value Objects).

Types Code Smells: Object-Orientation Abusers (Incorrect OOPs Usage)

Neglecting or improperly applying the principles of Object-Oriented Programming (OOPs).

Code Smell	Description	Refactoring Solution
Switch Statements	The excessive use of switch statements or a long series of if-else to distinguish types or behaviors.	🧱 Replace Conditional with Polymorphism (Use Subclasses or Interfaces).
Temporary Field	A field in a class that is only used within a few methods and is not needed for the object's entire lifespan.	🔨 Extract Class or Introduce Null Object.
Refused Bequest	A Subclass choosing not to use methods or fields from its Superclass (violating the inheritance contract).	➡️ Replace Inheritance with Delegation.

Types Code Smells: Change Preventers (Obstacles to Change)



Code designed in a way that requires changes in one part to ripple out, necessitating modifications in many other parts.

Code Smell	Description	Refactoring Solution
Divergent Change	If you need to change one capability (e.g., logging), you must change multiple methods within a single class.	✂️ Extract Class (Separate methods that change together).
Shotgun Surgery	If you need to change one capability, you must make small edits in many different classes.	➡️ Move Method or Move Field (Consolidate the changes into one class).
Parallel Inheritance Hierarchies	Adding a Subclass in one hierarchy requires adding a corresponding Subclass in another hierarchy.	🏗️ Move Method or Extract Class (Reduce co-dependency).

Types Code Smells: Dispensables (Things That Can Be "Thrown Away")

Unnecessary code or structures that add complexity without providing value.

Code Smell	Description	Refactoring Solution
Dead Code	Code that is never executed (e.g., a method that is never called).	🗑️ Remove Dead Code.
Duplicate Code	The same block of code appearing repeatedly in multiple locations or classes.	🛠️ Extract Method or Pull Up Method (Consolidate the duplicated code).
Speculative Generality	Creating capabilities "just in case" they might be needed in the future, without a current requirement.	💥 Remove Parameter or Collapse Hierarchy (Delete unnecessary complexity).

Types Code Smells: Couplers (Tight Linkages)

Issues related to excessive Dependency or tight Coupling between classes.

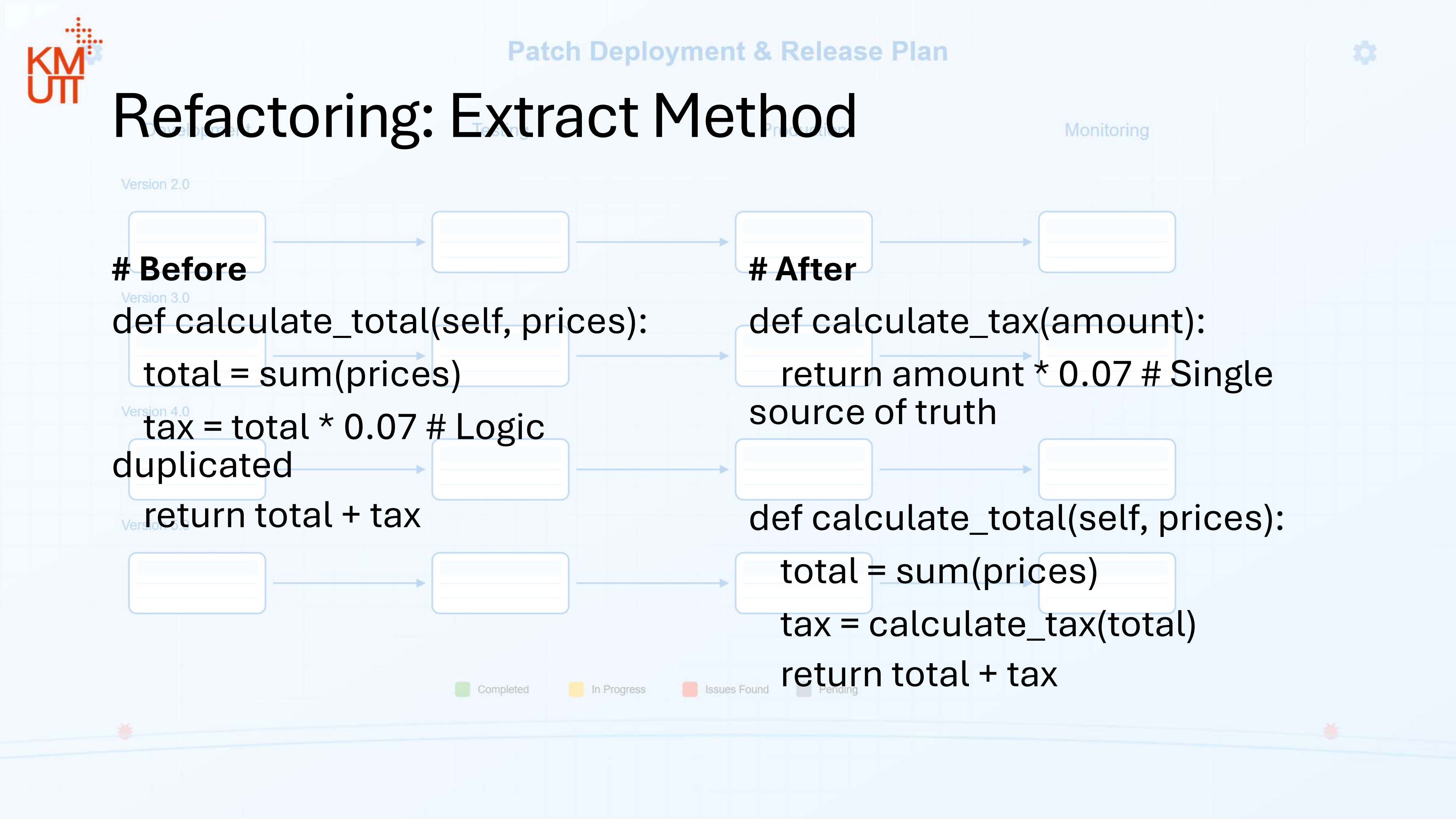
Code Smell	Description	Refactoring Solution
Feature Envy	A method in one class that appears to be "envious" of another class because it uses more data (fields/methods) from that other class than its own.	➡️ Move Method (Relocate the method to the appropriate class).
Inappropriate Intimacy	Two classes know too much about each other's internal details.	🔓 Change Bidirectional Association to Unidirectional or Move Method.
Message Chains	Long chains of method calls (e.g., obj.getA().getB().getC()) which violate the Law of Demeter.	✋ Hide Delegate (Conceal the internal linkages).Export to Sheets

Code Smells: Duplicated Code

```
# In Class A
def calculate_total(self, prices):
   total = sum(prices)
   tax = total * 0.07
   return total + tax

# In Class B (Duplicate!)
def compute_sum(self, costs):
   subtotal = sum(costs)
   vat = subtotal * 0.07
   return subtotal + vat
```

Refactoring: Extract Method



# Before

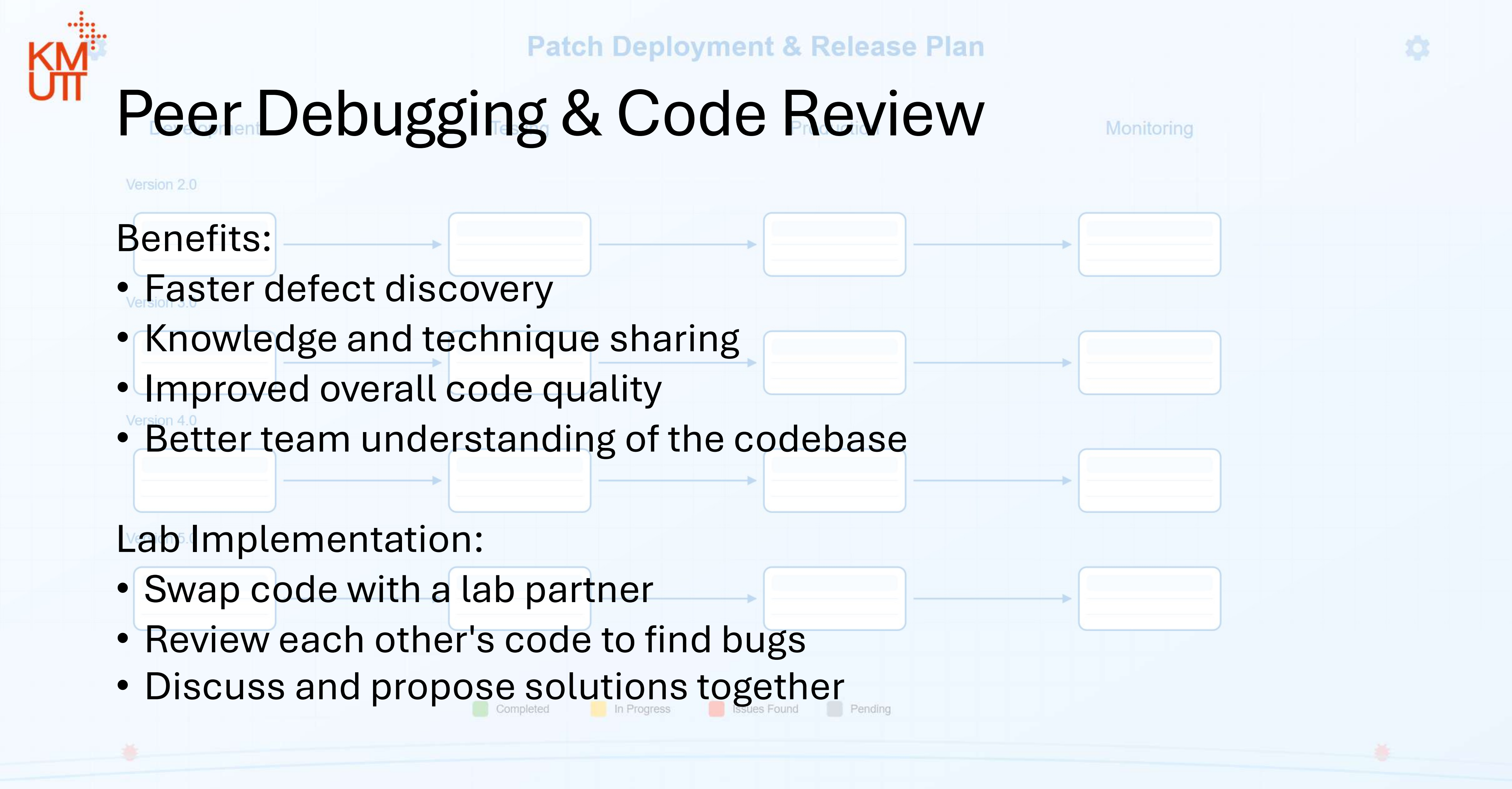
```
def calculate_total(self, prices):
    total = sum(prices)
    tax = total * 0.07 # Logic duplicated
    return total + tax
```

# After

```
def calculate_tax(amount):
    return amount * 0.07 # Single source of truth

def calculate_total(self, prices):
    total = sum(prices)
    tax = calculate_tax(total)
    return total + tax
```

Peer Debugging & Code Review

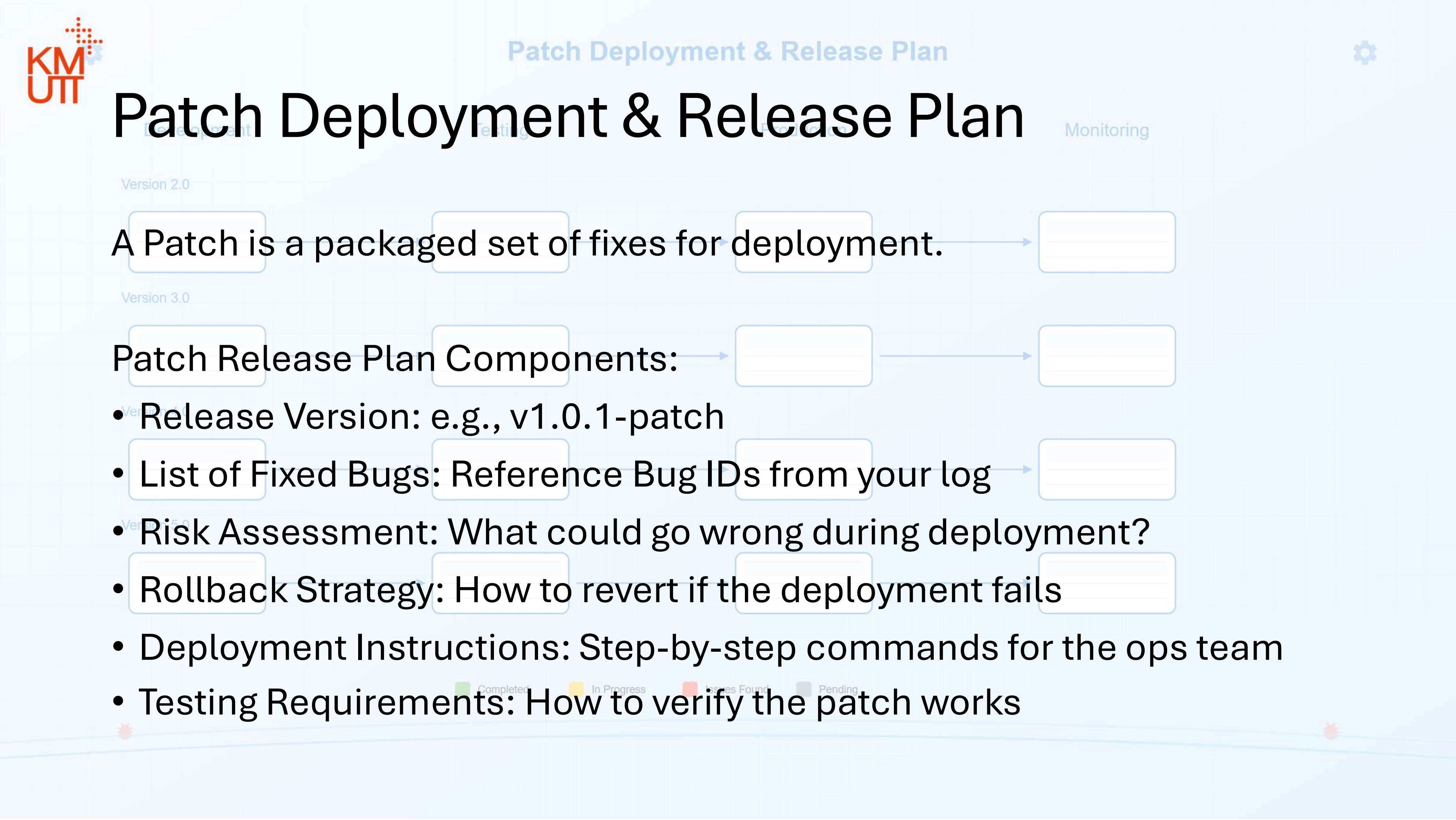


Benefits:

- Faster defect discovery
- Knowledge and technique sharing
- Improved overall code quality
- Better team understanding of the codebase

Lab Implementation:

- Swap code with a lab partner
- Review each other's code to find bugs
- Discuss and propose solutions together



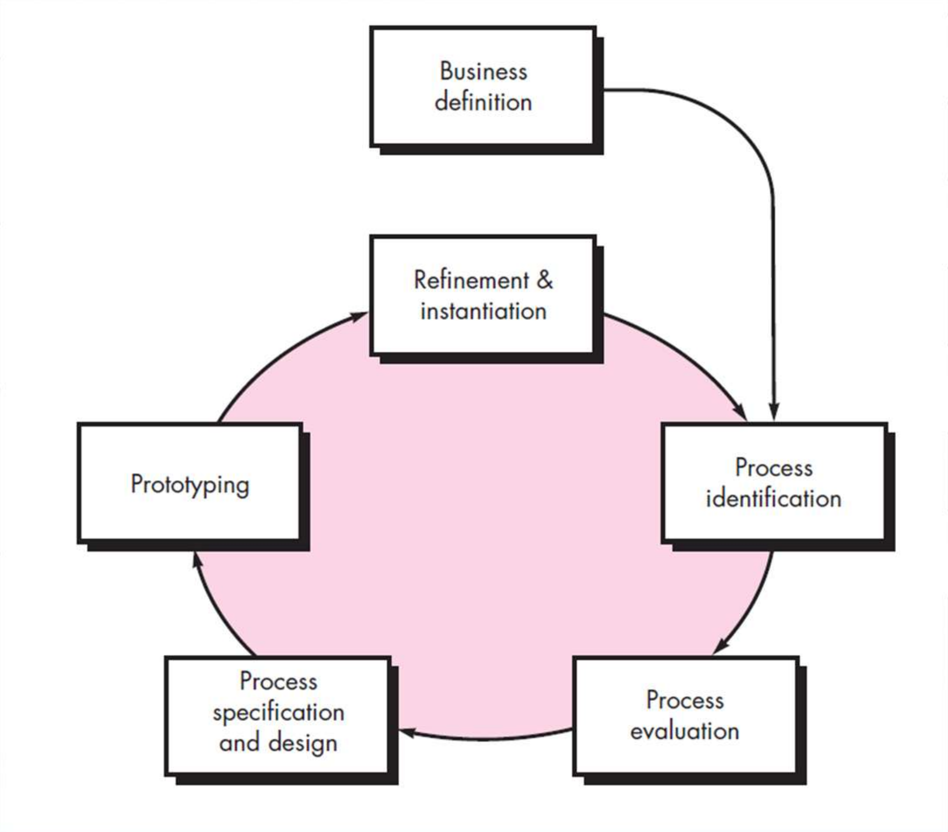
Patch Deployment & Release Plan

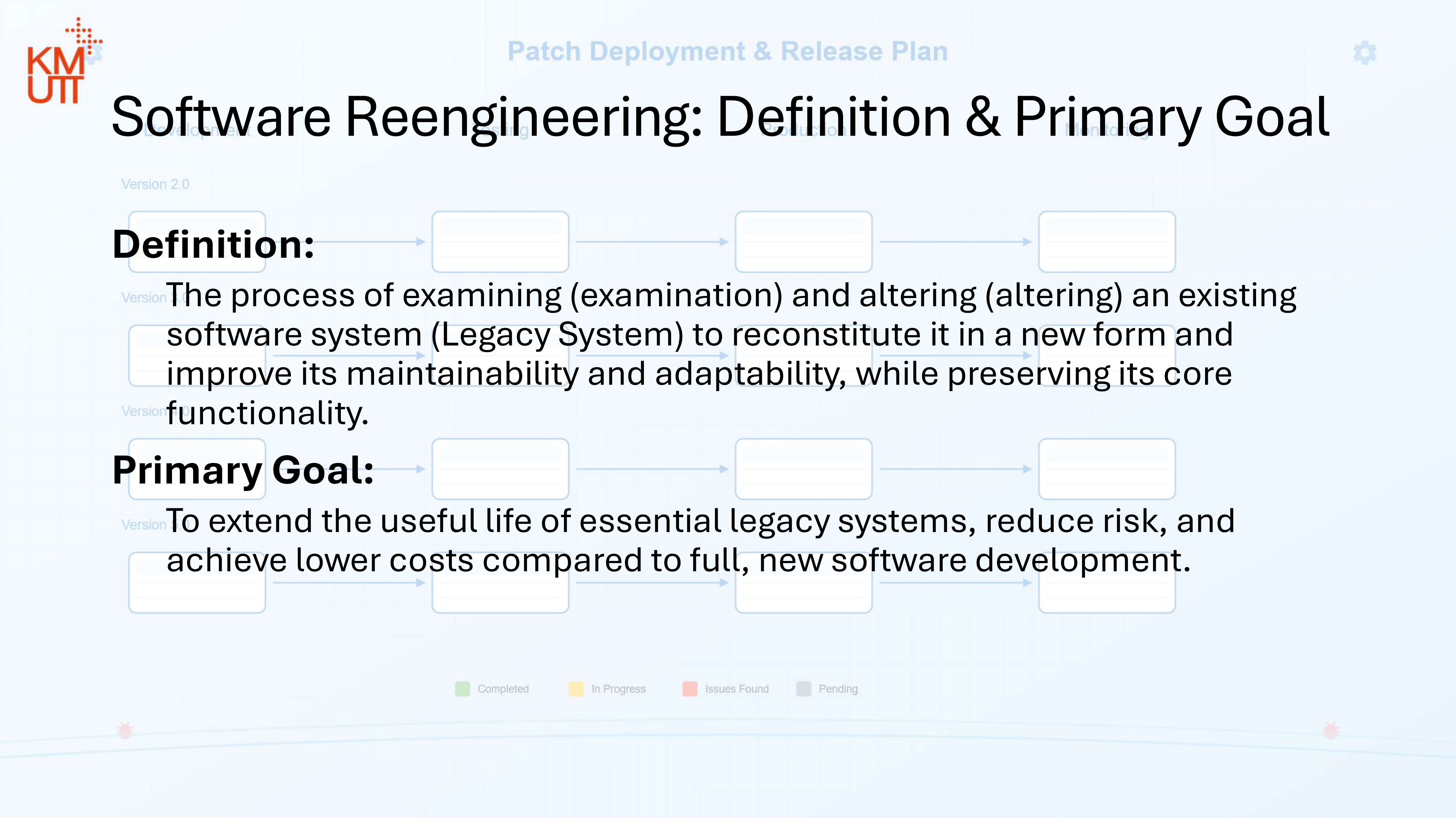
A Patch is a packaged set of fixes for deployment.

Patch Release Plan Components:

- Release Version: e.g., v1.0.1-patch
- List of Fixed Bugs: Reference Bug IDs from your log
- Risk Assessment: What could go wrong during deployment?
- Rollback Strategy: How to revert if the deployment fails
- Deployment Instructions: Step-by-step commands for the ops team
- Testing Requirements: How to verify the patch works

Business Process Reengineering Model





Software Reengineering: Definition & Primary Goal

Definition:

The process of examining (examination) and altering (altering) an existing software system (Legacy System) to reconstitute it in a new form and improve its maintainability and adaptability, while preserving its core functionality.

Primary Goal:

To extend the useful life of essential legacy systems, reduce risk, and achieve lower costs compared to full, new software development.

Why Reengineer? (The Need)

Challenges of Legacy Systems

- 🛑 Systems become difficult and costly to maintain (High maintenance burden)
- 🛑 Lack of documentation or outdated design materials
- 🛑 Performance degradation or inability to support evolving business requirements
- 🛑 Obsolete hardware/software support or underlying technology

Objectives of Reengineering

- ✅ Improve Maintainability and code structure (e.g., readability, testability)
- ✅ Migrate to New Technology (e.g., new language, platform, or architecture)
- ✅ Improve Quality of code and data structures
- ✅ Prepare for Functional Enhancement and future scalability

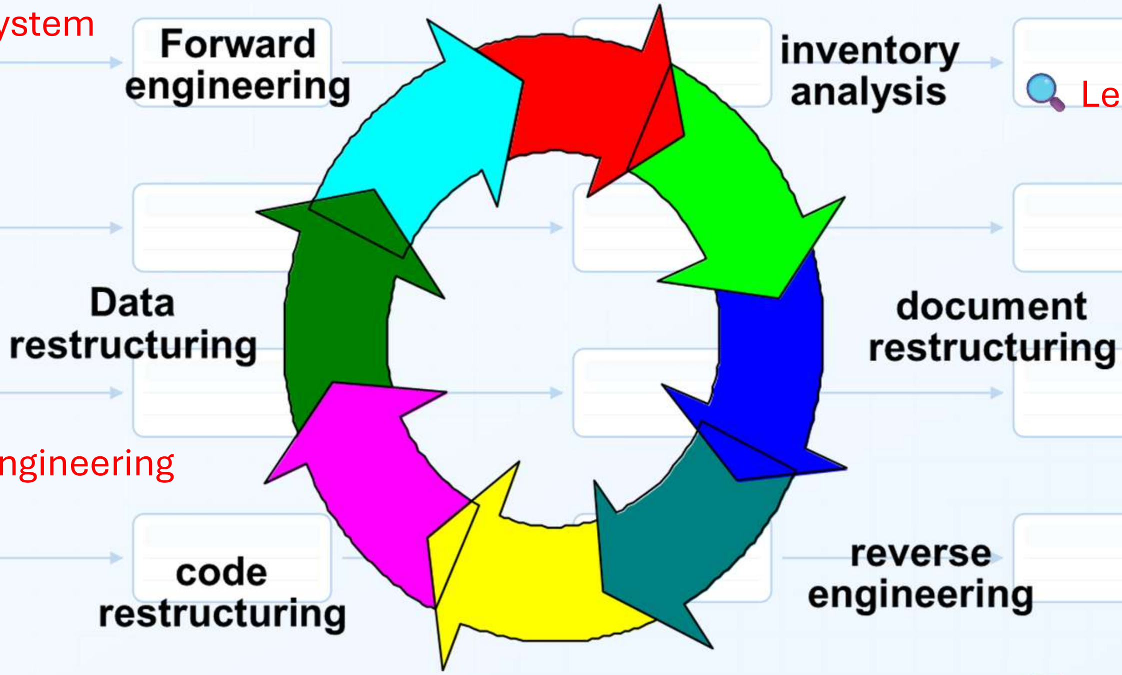
Key Advantages

- ⭐ Reduced Risk compared to starting from scratch
- ⭐ Lower Cost than complete system replacement

Software Reengineering Process Flow

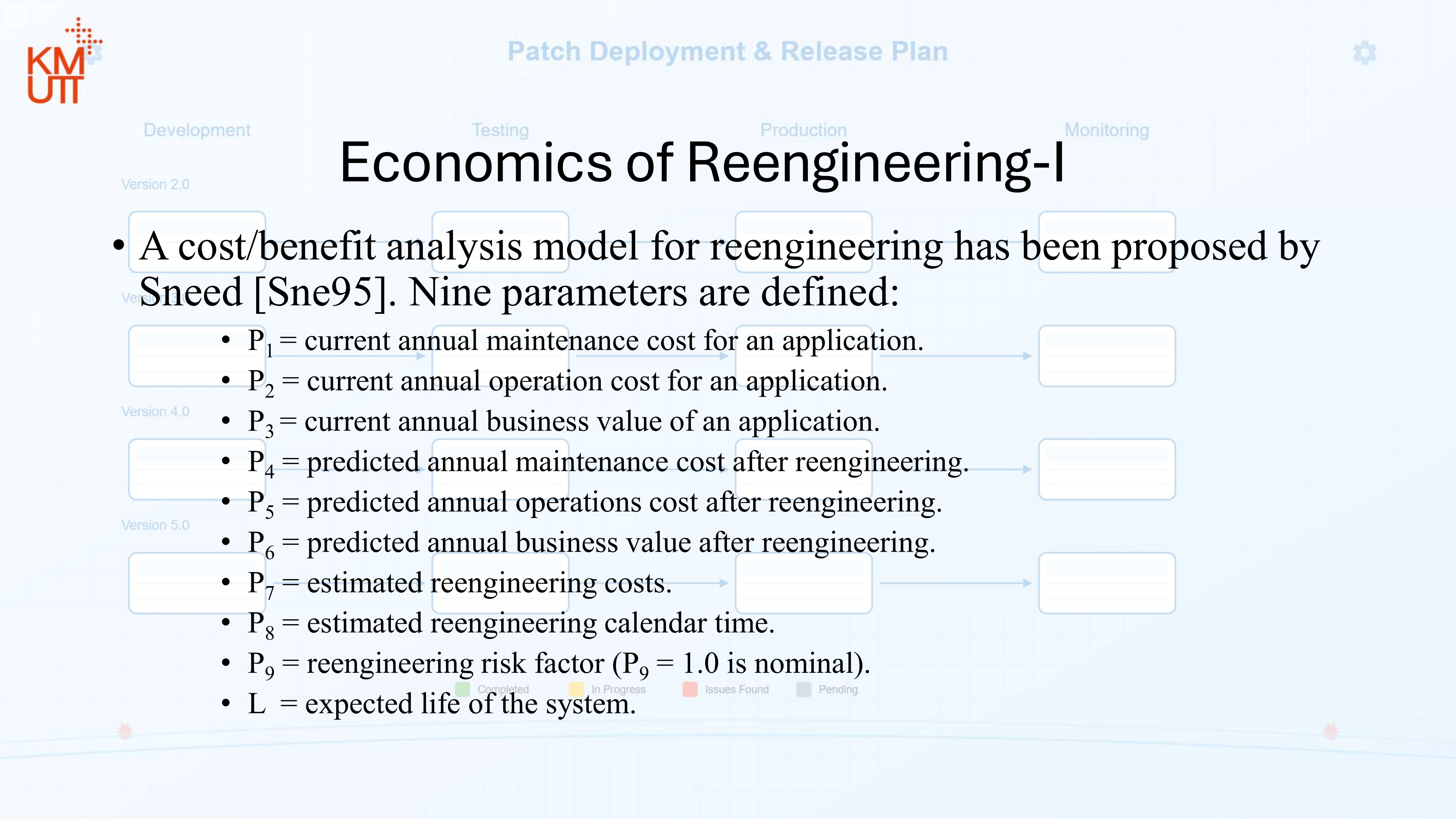
🚀 Generate Target System

🔍 Legacy System Assessment



🛠️ Program and Data Reengineering

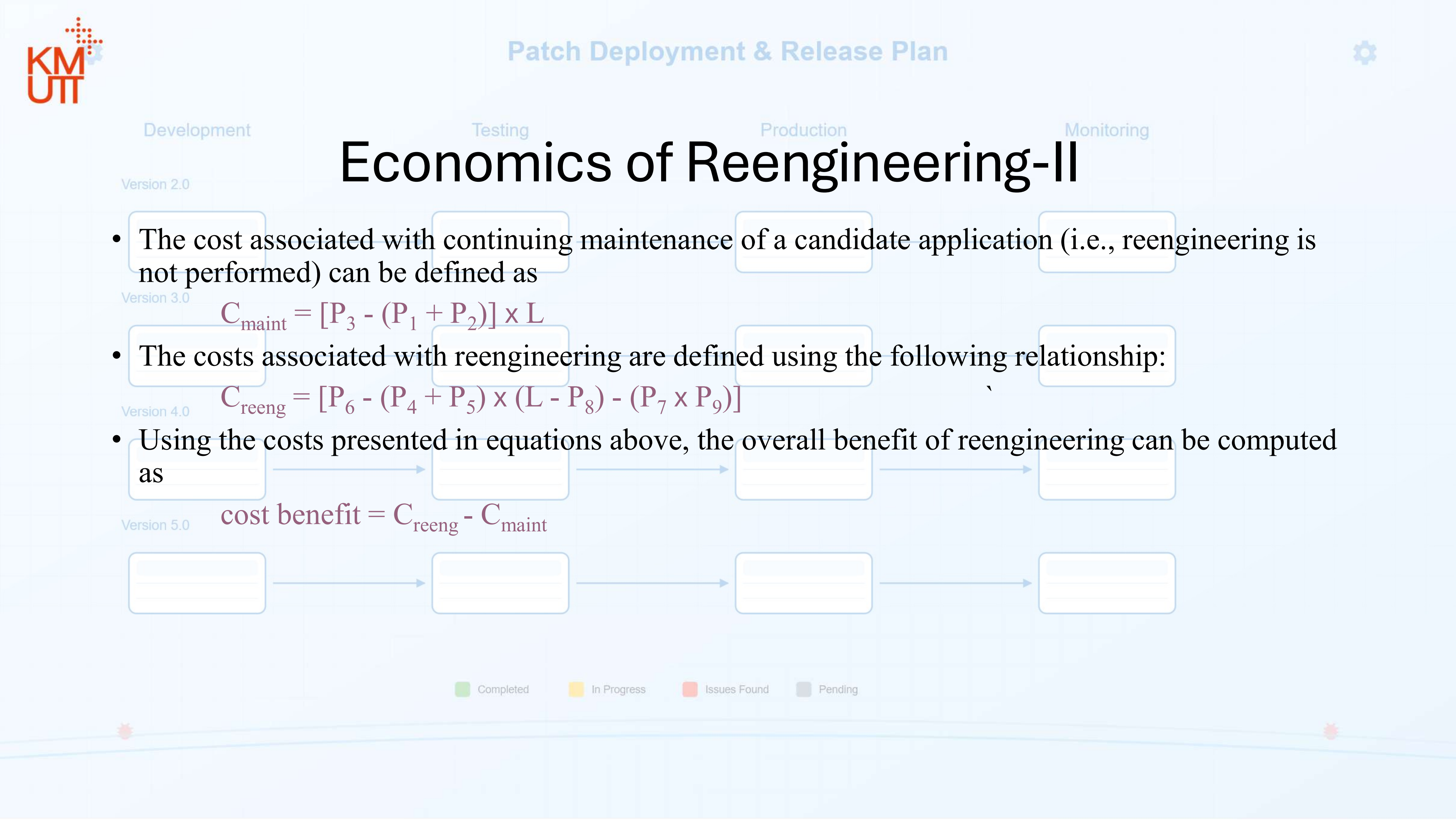
🏗️ Design Recovery



Economics of Reengineering-I

- A cost/benefit analysis model for reengineering has been proposed by Sneed [Sne95]. Nine parameters are defined:
    - P_1 = current annual maintenance cost for an application.
    - P_2 = current annual operation cost for an application.
    - P_3 = current annual business value of an application.
    - P_4 = predicted annual maintenance cost after reengineering.
    - P_5 = predicted annual operations cost after reengineering.
    - P_6 = predicted annual business value after reengineering.
    - P_7 = estimated reengineering costs.
    - P_8 = estimated reengineering calendar time.
    - P_9 = reengineering risk factor (P_9 = 1.0 is nominal).
    - L = expected life of the system.

Economics of Reengineering-II



- The cost associated with continuing maintenance of a candidate application (i.e., reengineering is not performed) can be defined as

$$C_{maint} = [P_3 - (P_1 + P_2)] \times L$$

- The costs associated with reengineering are defined using the following relationship:

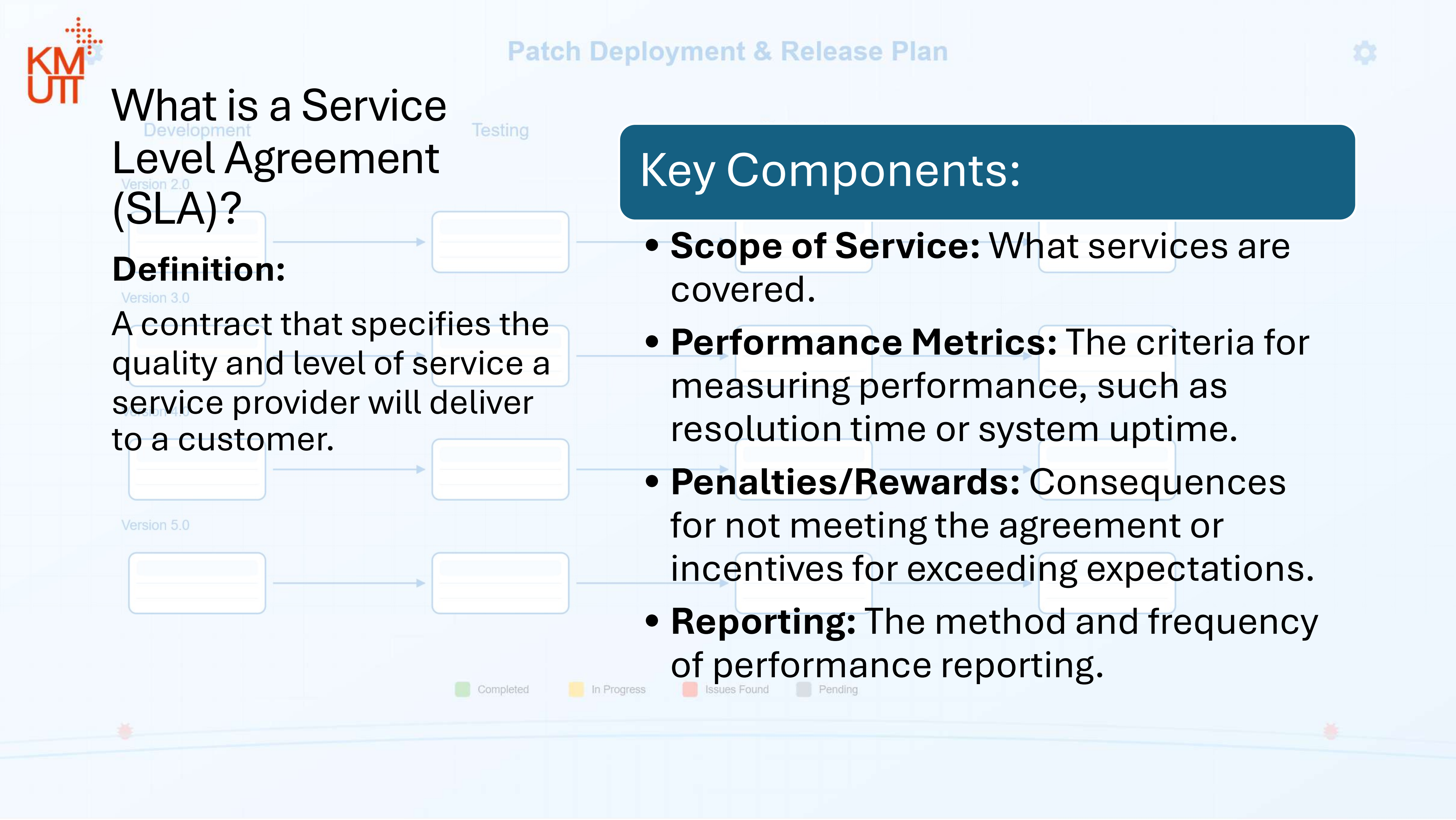
$$C_{reeng} = [P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9)]$$

`

- Using the costs presented in equations above, the overall benefit of reengineering can be computed as

$$\text{cost benefit} = C_{reeng} - C_{maint}$$

Service Level Agreement (SLA)



What is a Service Level Agreement (SLA)?

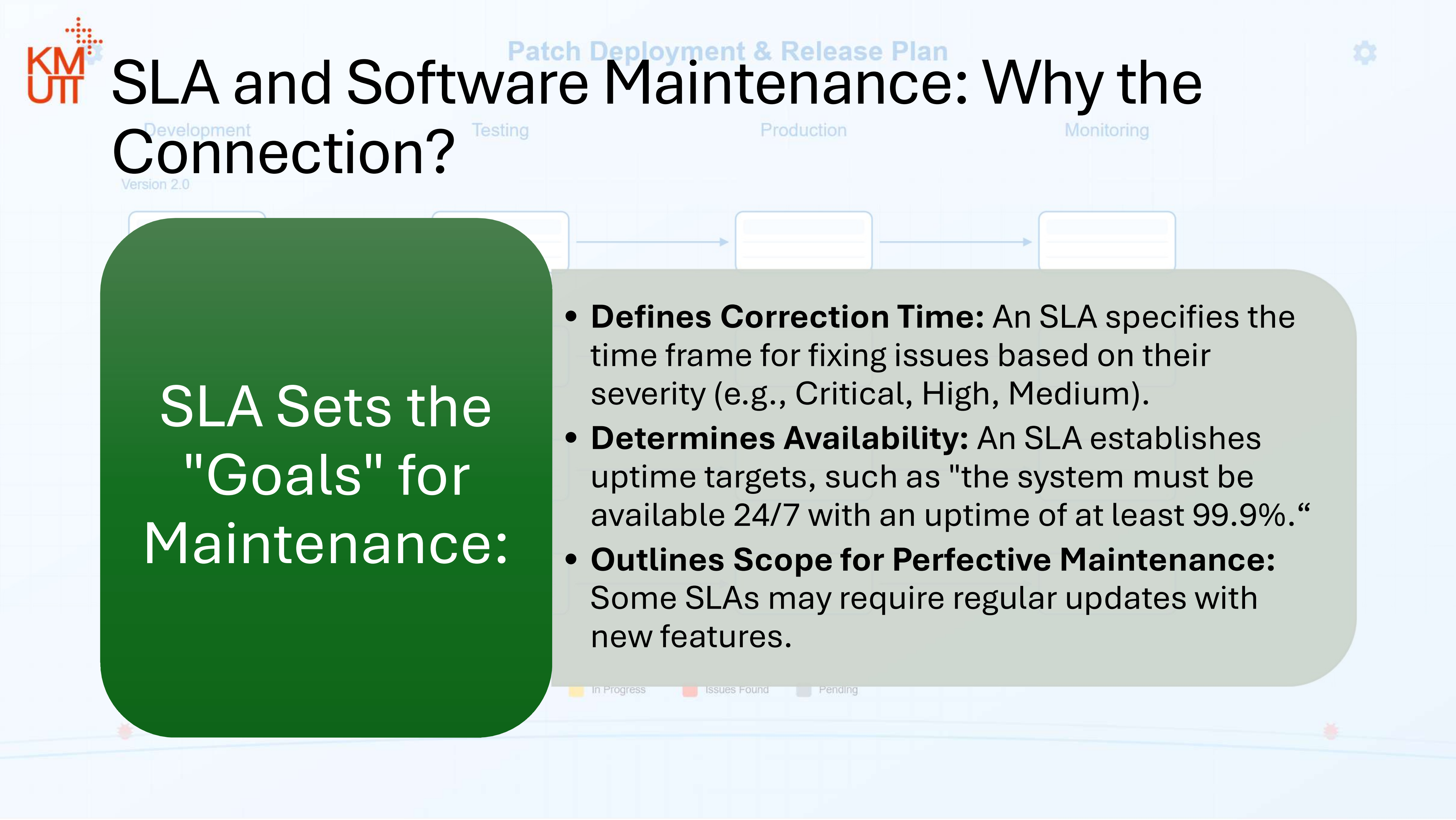
Definition:

A contract that specifies the quality and level of service a service provider will deliver to a customer.

Key Components:

- **Scope of Service:** What services are covered.
- **Performance Metrics:** The criteria for measuring performance, such as resolution time or system uptime.
- **Penalties/Rewards:** Consequences for not meeting the agreement or incentives for exceeding expectations.
- **Reporting:** The method and frequency of performance reporting.

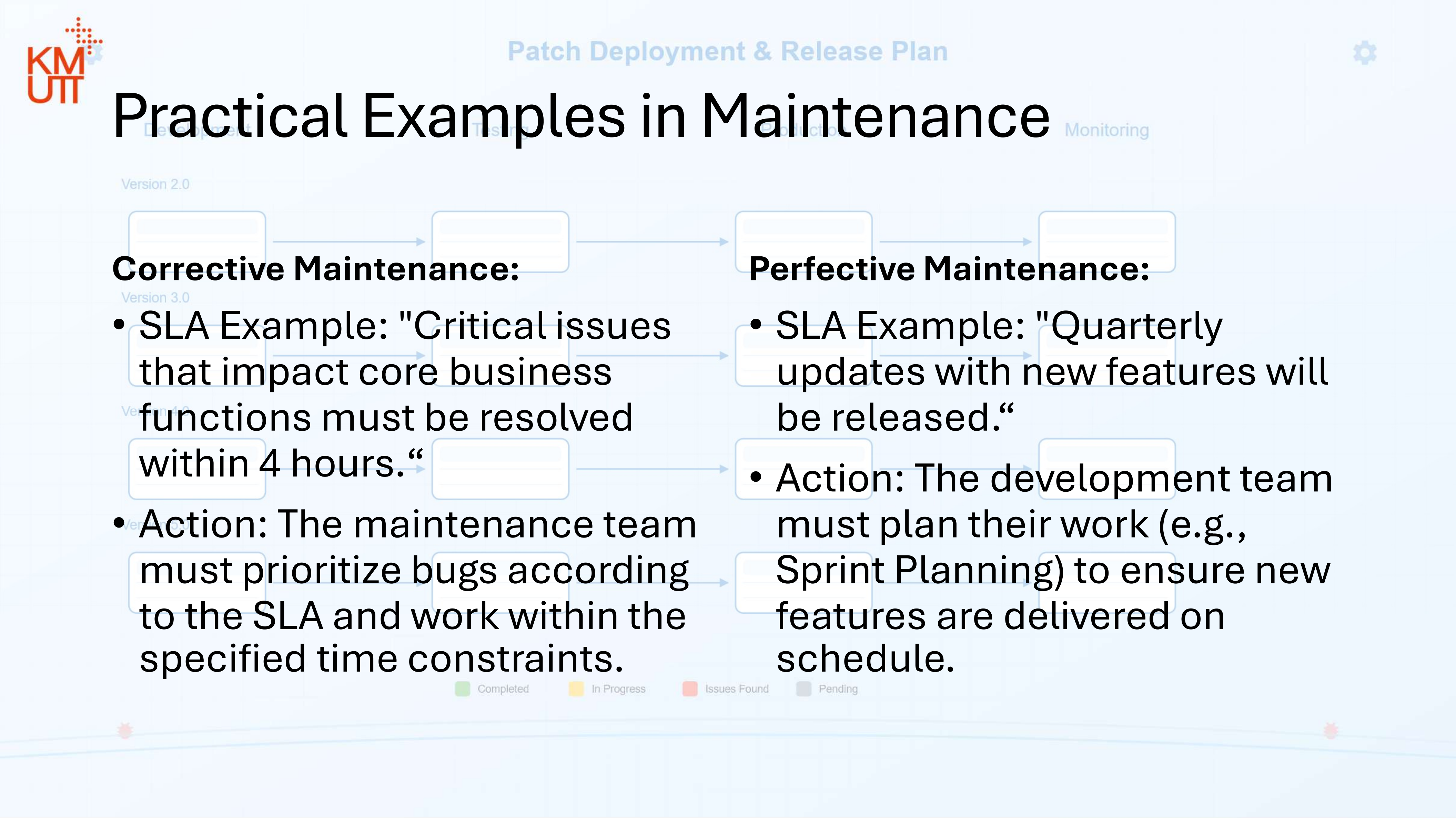
SLA and Software Maintenance: Why the Connection?



SLA Sets the "Goals" for Maintenance:

- **Defines Correction Time:** An SLA specifies the time frame for fixing issues based on their severity (e.g., Critical, High, Medium).
- **Determines Availability:** An SLA establishes uptime targets, such as "the system must be available 24/7 with an uptime of at least 99.9%.“
- **Outlines Scope for Perfective Maintenance:** Some SLAs may require regular updates with new features.

Practical Examples in Maintenance

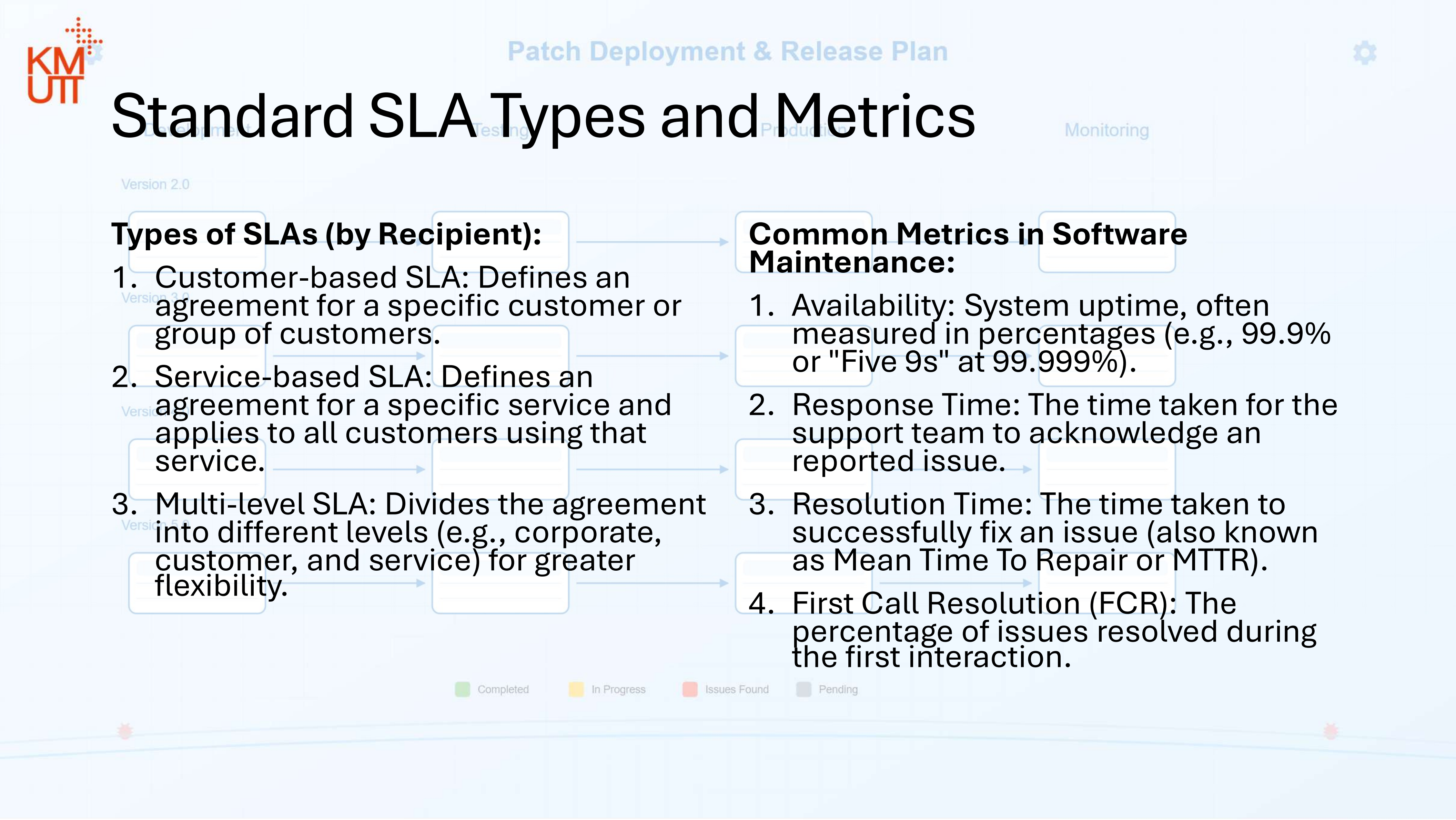


Corrective Maintenance:

- SLA Example: "Critical issues that impact core business functions must be resolved within 4 hours.“
- Action: The maintenance team must prioritize bugs according to the SLA and work within the specified time constraints.

Perfective Maintenance:

- SLA Example: "Quarterly updates with new features will be released.“
- Action: The development team must plan their work (e.g., Sprint Planning) to ensure new features are delivered on schedule.



Standard SLA Types and Metrics

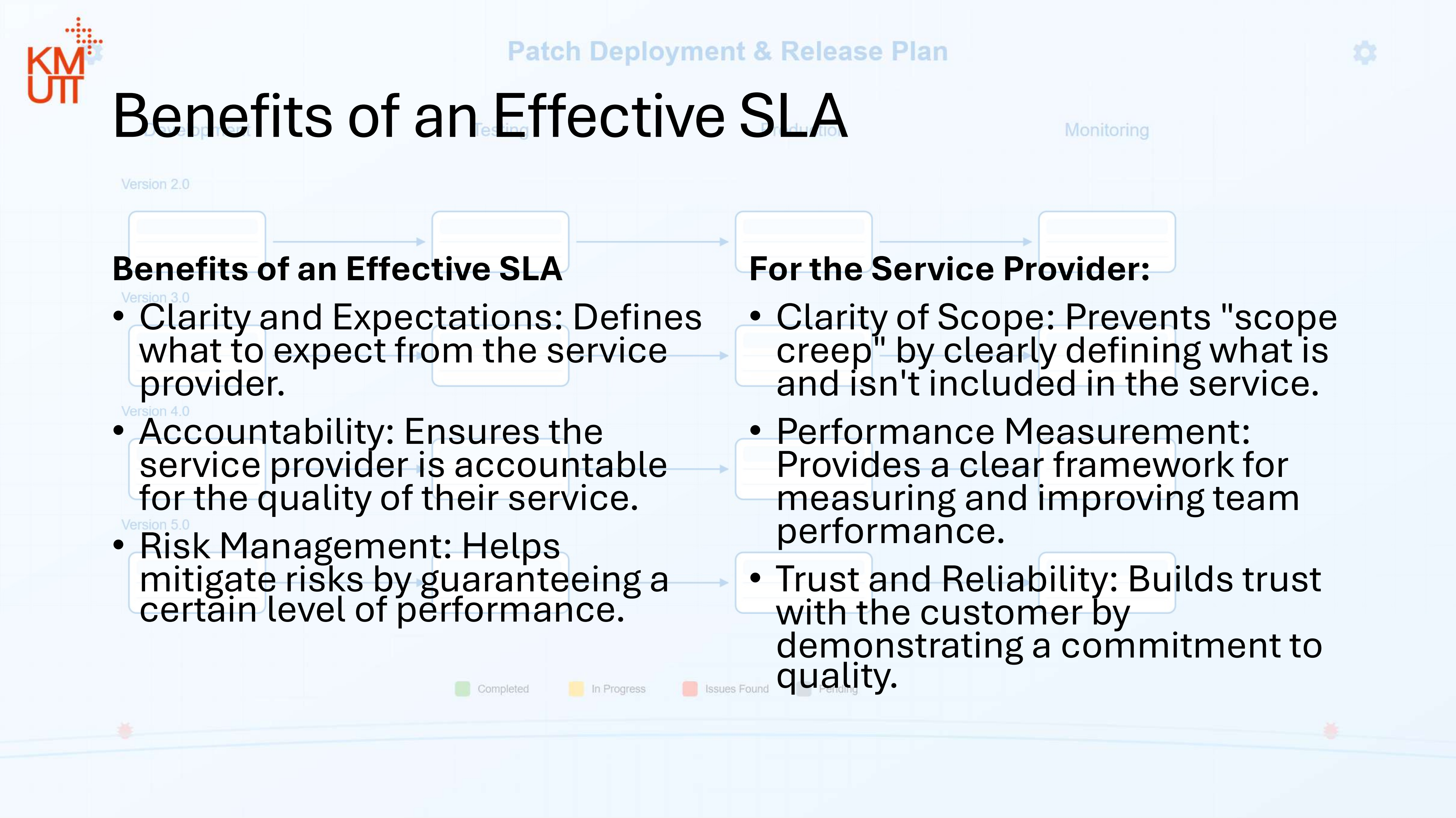
Types of SLAs (by Recipient):

1. Customer-based SLA: Defines an agreement for a specific customer or group of customers.
2. Service-based SLA: Defines an agreement for a specific service and applies to all customers using that service.
3. Multi-level SLA: Divides the agreement into different levels (e.g., corporate, customer, and service) for greater flexibility.

Common Metrics in Software Maintenance:

1. Availability: System uptime, often measured in percentages (e.g., 99.9% or "Five 9s" at 99.999%).
2. Response Time: The time taken for the support team to acknowledge an reported issue.
3. Resolution Time: The time taken to successfully fix an issue (also known as Mean Time To Repair or MTTR).
4. First Call Resolution (FCR): The percentage of issues resolved during the first interaction.

Benefits of an Effective SLA



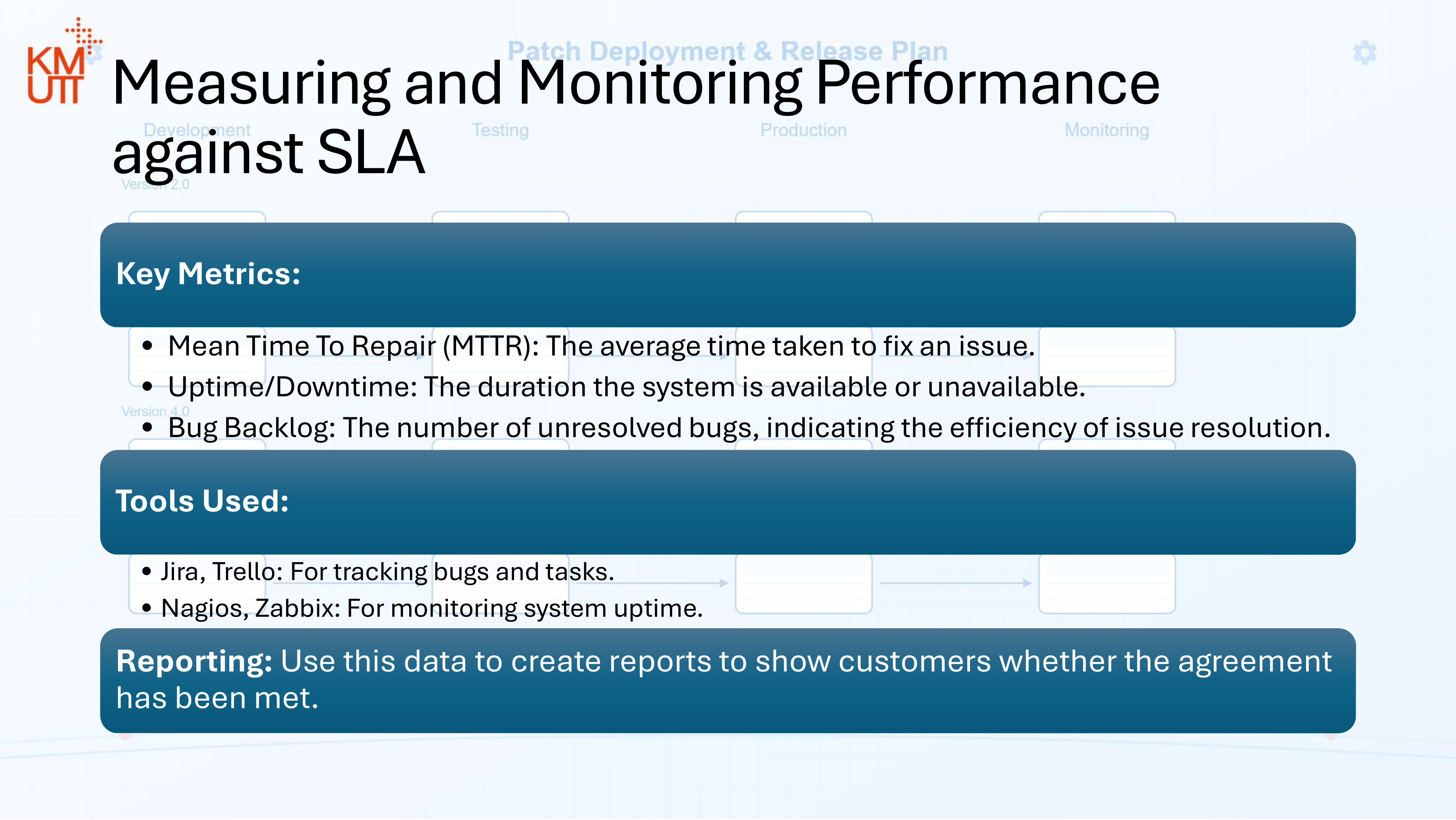
Benefits of an Effective SLA

- Clarity and Expectations: Defines what to expect from the service provider.
- Accountability: Ensures the service provider is accountable for the quality of their service.
- Risk Management: Helps mitigate risks by guaranteeing a certain level of performance.

For the Service Provider:

- Clarity of Scope: Prevents "scope creep" by clearly defining what is and isn't included in the service.
- Performance Measurement: Provides a clear framework for measuring and improving team performance.
- Trust and Reliability: Builds trust with the customer by demonstrating a commitment to quality.

Measuring and Monitoring Performance against SLA



Key Metrics:

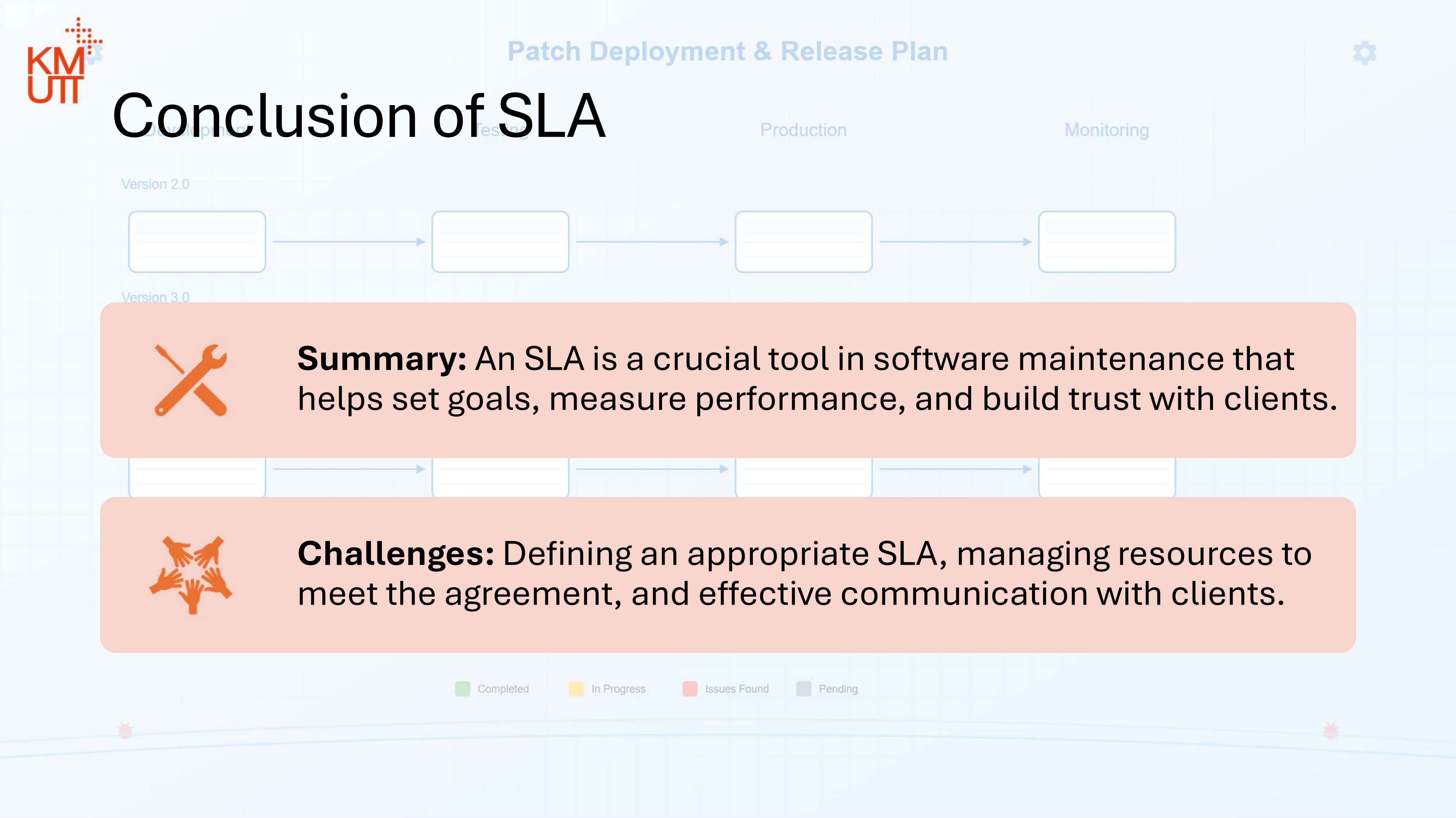
- Mean Time To Repair (MTTR): The average time taken to fix an issue.
- Uptime/Downtime: The duration the system is available or unavailable.
- Bug Backlog: The number of unresolved bugs, indicating the efficiency of issue resolution.

Tools Used:

- Jira, Trello: For tracking bugs and tasks.
- Nagios, Zabbix: For monitoring system uptime.

Reporting: Use this data to create reports to show customers whether the agreement has been met.

Conclusion of SLA

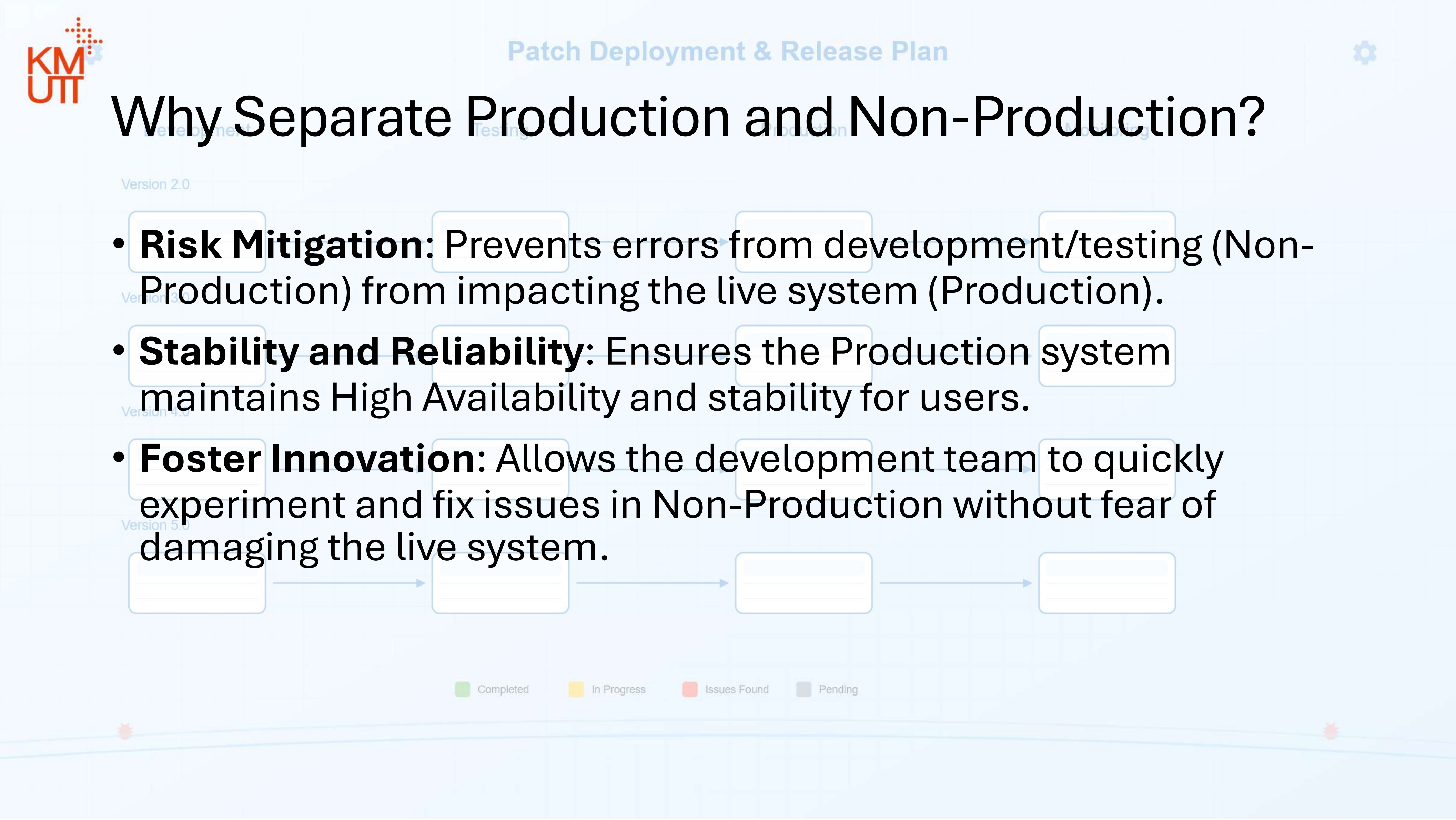


Summary: An SLA is a crucial tool in software maintenance that helps set goals, measure performance, and build trust with clients.

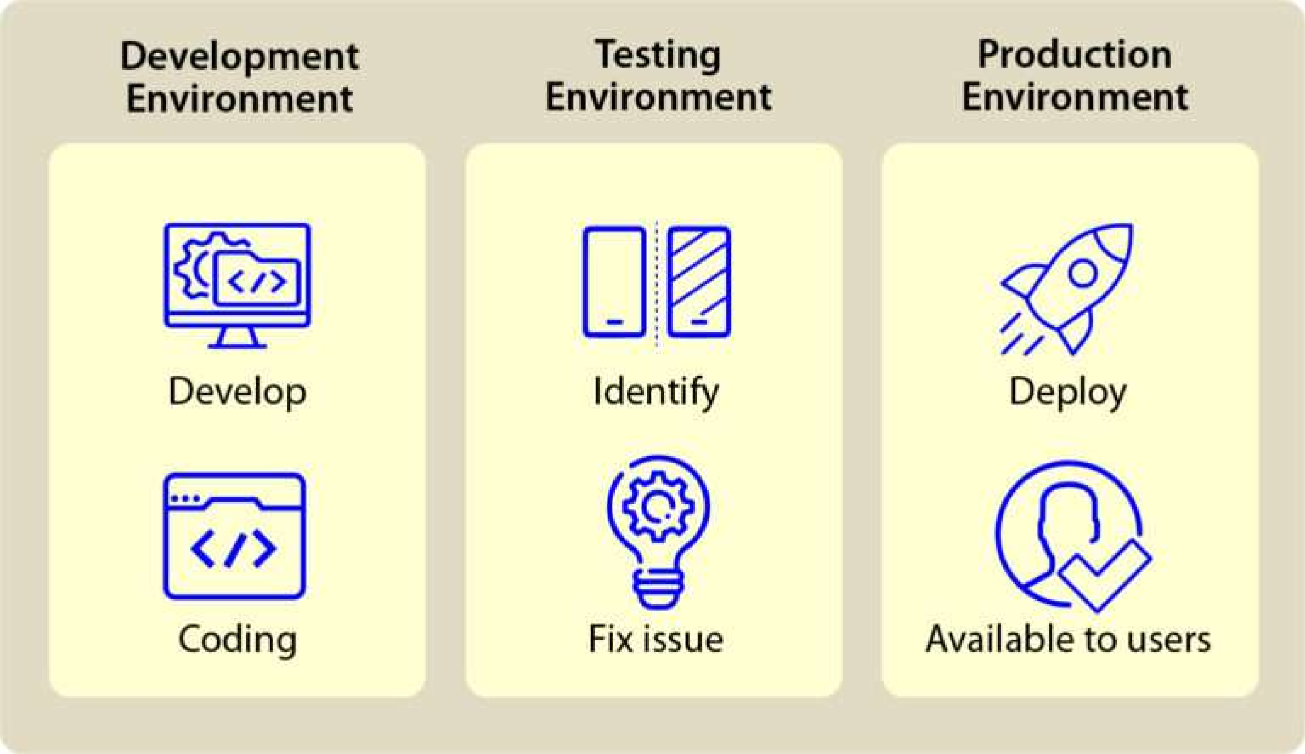
Challenges: Defining an appropriate SLA, managing resources to meet the agreement, and effective communication with clients.

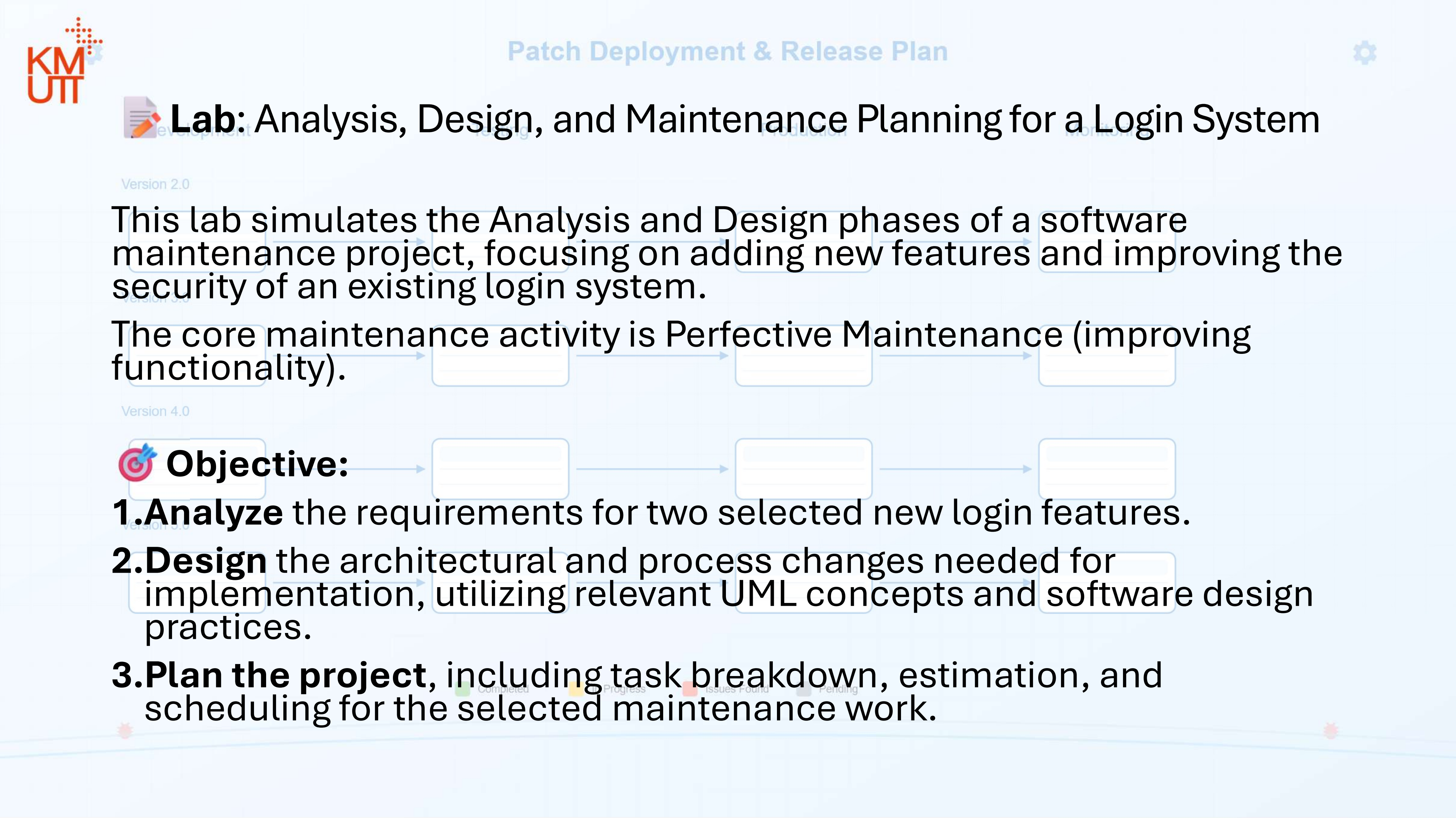
Why Separate Production and Non-Production?

- **Risk Mitigation**: Prevents errors from development/testing (Non-Production) from impacting the live system (Production).
- **Stability and Reliability**: Ensures the Production system maintains High Availability and stability for users.
- **Foster Innovation**: Allows the development team to quickly experiment and fix issues in Non-Production without fear of damaging the live system.



Managing IT Environments: Production vs. Non-Production





📝 **Lab**: Analysis, Design, and Maintenance Planning for a Login System

This lab simulates the Analysis and Design phases of a software maintenance project, focusing on adding new features and improving the security of an existing login system.

The core maintenance activity is Perfective Maintenance (improving functionality).

🎯 Objective:

1. **Analyze** the requirements for two selected new login features.
2. **Design** the architectural and process changes needed for implementation, utilizing relevant UML concepts and software design practices.
3. **Plan the project**, including task breakdown, estimation, and scheduling for the selected maintenance work.

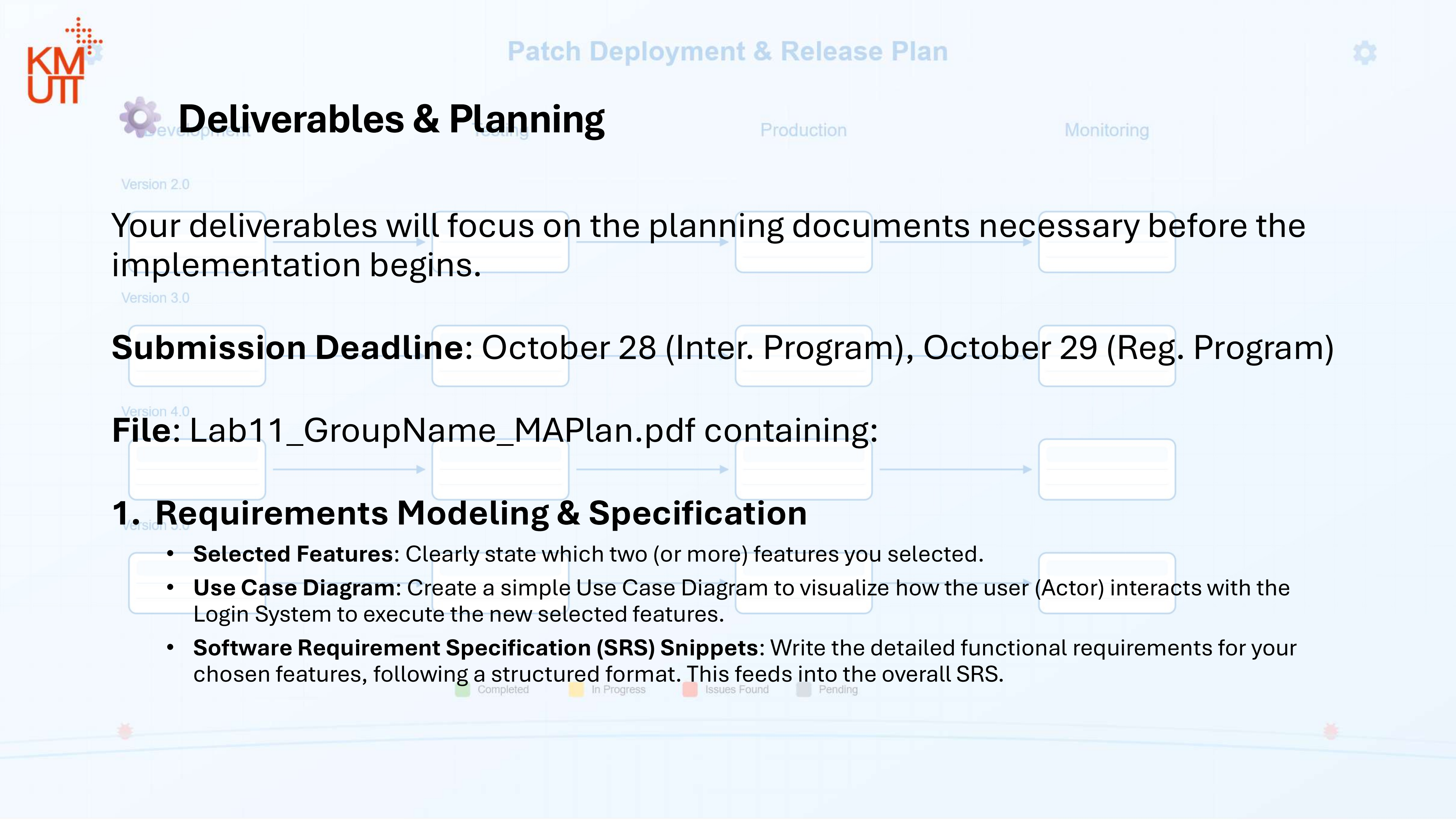
🛠️ **The Task**: Login System Enhancement

You are to select and implement at least two (2) new features for the existing login system from the list below.

Feature Options (Select at least 2)

Feature	Description
1. Local User Login	(Core Feature for Analysis) Implement standard login capability for local user accounts (username/email and password validation against the system's database). You must design around this feature.
2. Two-Factor Authentication (2FA)	Implement a secondary verification step (e.g., via a code sent to a mobile device or email) after the initial password entry.
3. Enhanced Password Policy	Enforce a stronger password policy: password must be at least 8 characters long and contain at least one Capital Letter, one Lowercase Letter, one Special Character, and one Number.
4. Self-Service Password Reset	Allow users who forgot their password to request a password change themselves, using the registered email or phone number for identity verification.
5. Session Timeout	Set the user's active system session to expire 2 hours after the session becomes inactive (no user activity)

This lab exercise utilizes concepts from Requirements Engineering (Weeks 2-3), Architecture & Design (Weeks 4-5), and Project Planning (Week 7), and culminates in a focus on the Maintenance Phase (Week 11).



Deliverables & Planning

Your deliverables will focus on the planning documents necessary before the implementation begins.

Submission Deadline: October 28 (Inter. Program), October 29 (Reg. Program)

File: Lab11_GroupName_MAPlan.pdf containing:

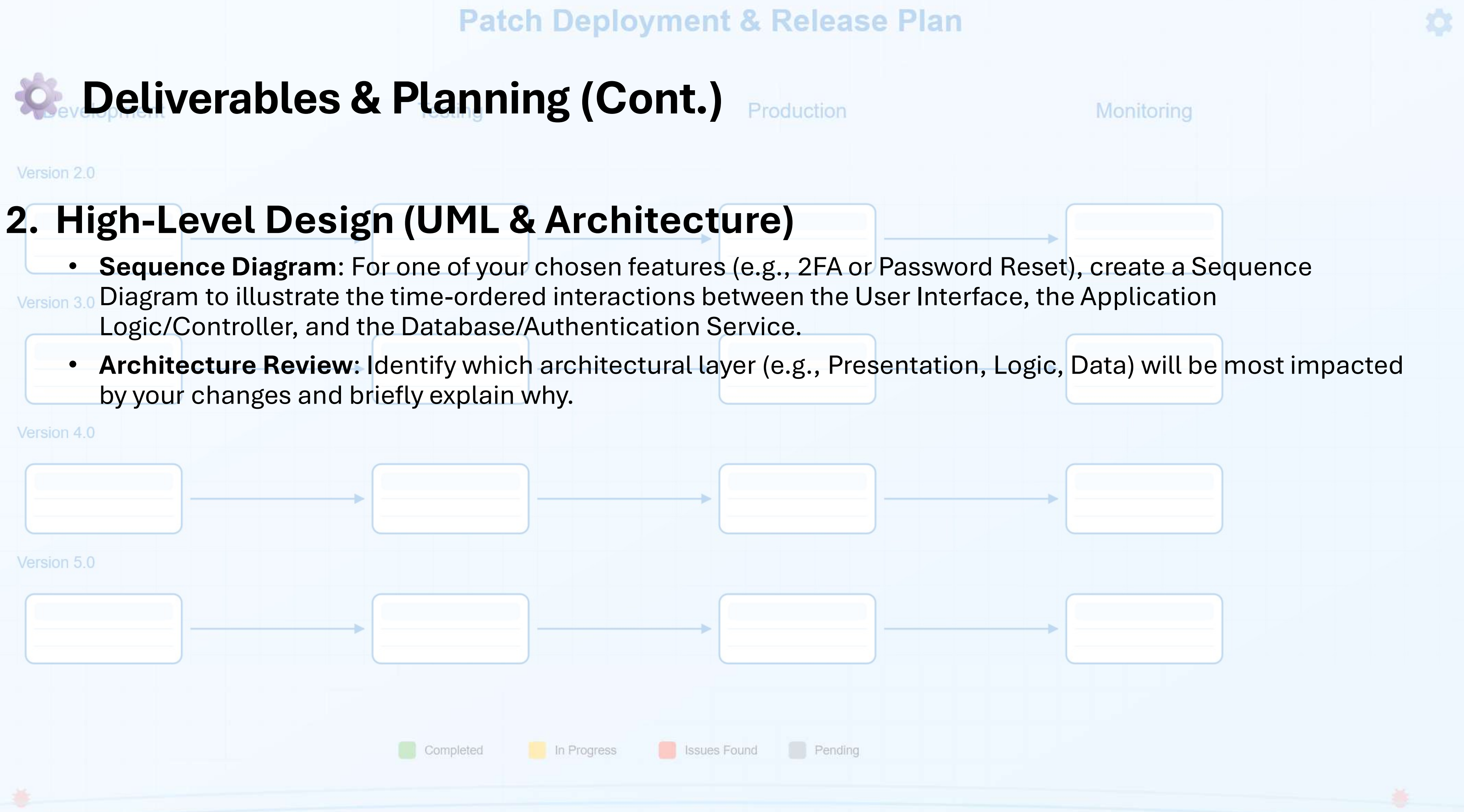
1. Requirements Modeling & Specification

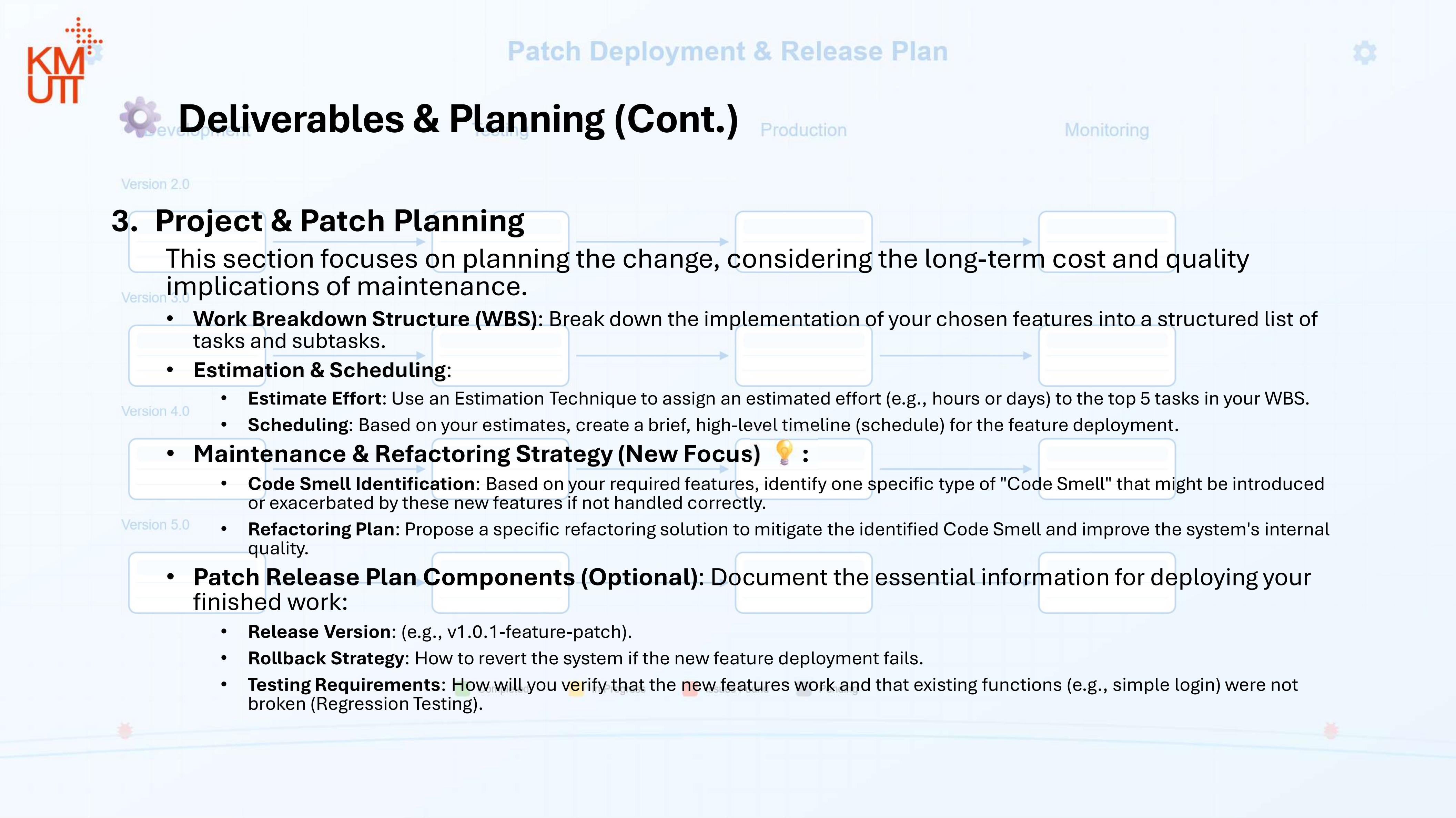
- **Selected Features**: Clearly state which two (or more) features you selected.
- **Use Case Diagram**: Create a simple Use Case Diagram to visualize how the user (Actor) interacts with the Login System to execute the new selected features.
- **Software Requirement Specification (SRS) Snippets**: Write the detailed functional requirements for your chosen features, following a structured format. This feeds into the overall SRS.

Deliverables & Planning (Cont.)

2. High-Level Design (UML & Architecture)

- **Sequence Diagram**: For one of your chosen features (e.g., 2FA or Password Reset), create a Sequence Diagram to illustrate the time-ordered interactions between the User Interface, the Application Logic/Controller, and the Database/Authentication Service.
- **Architecture Review:** Identify which architectural layer (e.g., Presentation, Logic, Data) will be most impacted by your changes and briefly explain why.





Deliverables & Planning (Cont.)

3. Project & Patch Planning

This section focuses on planning the change, considering the long-term cost and quality implications of maintenance.

- **Work Breakdown Structure (WBS)**: Break down the implementation of your chosen features into a structured list of tasks and subtasks.
- **Estimation & Scheduling**:
  - **Estimate Effort**: Use an Estimation Technique to assign an estimated effort (e.g., hours or days) to the top 5 tasks in your WBS.
  - **Scheduling**: Based on your estimates, create a brief, high-level timeline (schedule) for the feature deployment.
- **Maintenance & Refactoring Strategy (New Focus)** 💡 **:**
  - **Code Smell Identification**: Based on your required features, identify one specific type of "Code Smell" that might be introduced or exacerbated by these new features if not handled correctly.
  - **Refactoring Plan**: Propose a specific refactoring solution to mitigate the identified Code Smell and improve the system's internal quality.
- **Patch Release Plan Components (Optional)**: Document the essential information for deploying your finished work:
  - **Release Version**: (e.g., v1.0.1-feature-patch).
  - **Rollback Strategy**: How to revert the system if the new feature deployment fails.
  - **Testing Requirements**: How will you verify that the new features work and that existing functions (e.g., simple login) were not broken (Regression Testing).

Q&A / References

References:

- Sommerville, Ian. Software Engineering, 10th Global Edition. Chapter 9.
- Pressman, Roger S. Software Engineering: A Practitioner's Approach, 7th Edition. Chapter 29.
- Fowler, Martin. Refactoring: Improving the Design of Existing Code.
- Code Smell: SourceMaking.com, Refactoring.Guru

Questions?