



MASTERARBEIT | MASTER'S THESIS

Titel | Title

Expedite Lane in Kanban: How urgent-work policies affect
predictability

verfasst von | submitted by

Yasin Sivrikaya BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2026

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 915

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Betriebswirtschaft

Betreut von | Supervisor:

Dr. David Winkelmann B.Sc. M.Sc.

Abstract

Predictability is a main quality of Kanban, but is influenced by various factors. One of which is the continuous interruption through urgent work. In practice, there are multiple policies for handling urgent work, but it is not clear how each of them affects predictability when applied to real Jira data. This thesis looks at how four such policies affect the predictability of normal work in a Kanban system. The policies compared are Priority Jumping, Preemption, Reserved Capacity, and Strict Expedite WIP Limit.

The study combines two parts, the first of which analyzes real Jira data from the SPARK project (2015–2018, with 19,492 normal and 1,910 urgent arrivals). These data are then used as input for a discrete-event simulation that compares the four policies under the same conditions. Predictability of normal work is measured through Lead Time percentiles (P50, P85, P95) and how much these vary between simulation runs. Urgent work performance is measured as the share of urgent items completed within a specific threshold. The baseline was done with a capacity of 1000. To check the robustness of the results, the simulation was also repeated with varying capacities (800 and 1,200) and with a capped service-time input (Trim995).

The results showed that no single policy was the best in all situations. Strict Expedite WIP Limit had the lowest Lead Times for normal work, but the worst results for urgent work. Preemption gave the best urgent-work performance. Priority Jumping was the most balanced policy and showed the lowest spread for normal work. Reserved Capacity performed the worst for normal work and did not give a clear benefit for urgent work. The findings suggest that urgent-work policies should not be treated as exceptions, but as clear, measurable rules of the system. These policies decide how delay and predictability are spread across the workflow.

Kurzfassung

Kanban ist eines von vielen Frameworks, die in der IT eingesetzt werden. Je nach Anwendungsgebiet rücken dabei unterschiedliche Stärken in den Vordergrund, bei Kanban ist es die Vorhersagbarkeit. Diese wird in der Praxis jedoch von verschiedenen Faktoren beeinflusst. Einer davon ist die Unterbrechung regulärer Arbeit durch dringliche Arbeit. Um mit solchen Unterbrechungen umzugehen, kommen in der Praxis verschiedene Richtlinien zum Einsatz. Wie sich diese einzelnen Richtlinien auf die Vorhersagbarkeit auswirken, wenn echte Jira-Daten zugrunde gelegt werden, ist bisher unklar. Diese Masterarbeit untersucht vier solcher Richtlinien und ihren Einfluss auf die Vorhersagbarkeit in einem Kanban-System: Priority Jumping, Preemption, Reserved Capacity und Strict Expedite WIP Limit.

Um die Simulation durchführen zu können, war es zunächst notwendig, echte Jira-Daten aus dem SPARK-Projekt (2015–2018, mit 19.492 regulären und 1.910 dringlichen Arbeitseingängen) zu analysieren. Diese Daten dienten anschließend als Input für eine ereignisdiskrete Simulation, in der die vier Richtlinien unter denselben Bedingungen verglichen werden. Die Vorhersagbarkeit regulärer Arbeit wird über Lead-Time-Perzentile (P50, P85, P95) gemessen sowie über deren Schwankung zwischen den Simulationsläufen. Die Performance dringlicher Arbeit wird über den Anteil dringlicher Arbeitseinheiten gemessen, die innerhalb eines bestimmten Schwellenwerts abgeschlossen werden. Um die Stabilität der Ergebnisse zu prüfen, wurde die Simulation zusätzlich mit abweichenden Kapazitäten (800 und 1.200) sowie mit einem gekappten Service-Time-Input wiederholt.

Keine der vier Richtlinien erweist sich in allen Situationen als die beste. Strict Expedite WIP Limit erreichte zwar die niedrigsten Lead Times für reguläre Arbeit, schnitt bei dringlicher Arbeit jedoch am schlechtesten ab. Preemption lieferte die beste Performance für dringliche Arbeit. Priority Jumping erwies sich als ausgewogenste Richtlinie und zeigte den geringsten Spread bei regulärer Arbeit. Reserved Capacity schnitt bei regulärer Arbeit am schlechtesten ab und lieferte keinen klaren Mehrwert für dringliche Arbeit. Diese Ergebnisse legen nahe, dass Richtlinien für dringliche Arbeit nicht als Ausnahmen behandelt, sondern als klare, messbare Regeln des Systems verstanden werden sollten. Denn genau diese Richtlinien entscheiden, wie sich Verzögerung und Vorhersagbarkeit über den Prozess verteilen.

Contents

Abstract	i
Kurzfassung	ii
List of Figures	vi
List of Tables	vii
List of Listings	viii
1 Introduction	1
1.1 Research questions	2
2 Theoretical Background	4
2.1 Predictability	5
2.1.1 Variability and averages	6
2.1.2 Measuring and steering	7
2.2 Kanban	7
2.2.1 Kanban as a queueing system	8
2.2.2 The six practices	9
2.2.3 Empirical evidence	10
2.2.4 Kanban in the service context	11
2.3 Flow metrics	11
2.3.1 The four central flow metrics	12
2.3.2 Little's Law	13
2.3.3 Averages	13
2.3.4 Service Level Expectation (SLE)	15
2.3.5 Scope and relevance for this thesis	15
2.4 Urgent work	16
2.4.1 Frequency and patterns of priority changes	16
2.4.2 Cognitive costs of interruptions	17
2.4.3 Preemption and priority policies	18
2.4.4 Flow debt	19
2.4.5 Implications for this thesis: urgent work	20
2.5 Classes of service	21
2.5.1 Definition and operational core	21
2.5.2 Practical implementation on the board	21
2.5.3 The four archetypes	22

Contents

2.5.4	Simulation based evidence	23
2.5.5	The four expedite policies	25
2.5.6	Limits of the concept	28
2.5.7	Implications for this thesis: classes of service	28
3	Method	30
3.1	Data Preparation	30
3.1.1	Dataset	31
3.1.2	Data cleaning and classification	31
3.1.3	Completion and Lead Time	32
3.1.4	Arrivals	33
3.1.5	Service time	34
3.1.6	Summary of simulation inputs	35
3.2	Simulation Model	35
3.2.1	Agents and components	35
3.2.2	Arrival handling	36
3.2.3	Dispatching and policy logic	36
3.2.4	Service and completion	36
3.2.5	Termination and outputs	37
3.2.6	Model assumptions	37
3.3	Parameterization	38
3.3.1	Capacity	38
3.3.2	Policy parameters	39
3.3.3	Replications and termination	39
4	Results	41
4.1	Baseline scenario	42
4.1.1	Normal work	42
4.1.2	Urgent work	43
4.1.3	Run-level summary	45
4.2	Robustness scenarios	46
4.2.1	800 servers	46
4.2.2	1,200 servers	49
4.2.3	Summary across capacity levels	51
4.3	Service time sensitivity	54
5	Discussion	57
5.1	Interpretation of the main findings	57
5.2	Answering the research questions	60
5.3	Relation to theoretical background	61
5.4	Methodological strengths	62
5.5	Limitations	63
5.6	Implications	65

Contents

6 Conclusion	67
6.1 Future research	68
References	69
A Appendix	71
A.0 Reproducibility: SPARK Jira Log Preparation	71
A.1 README	71
A.2 Entry point: <code>run_repro.sh</code>	72
A.3 Data preparation pipeline: <code>repro_pipeline.R</code>	72
A.4 Validation script: <code>validate.R</code>	78
A.5 Simulation model documentation	84

List of Figures

2.1	Simplified Kanban board	9
2.2	Cycle time scatterplot with percentile lines. Own illustration, based on Vacanti (2025).	14
2.3	Comparison of the four simulation scenarios. Own illustration, based on Vacanti (2025).	24
2.4	The four expedite policies along two structural dimensions. Own illustration, based on Vacanti (2025) and Harchol-Balter (2013).	27
3.1	Simulation model flow (AnyLogic).	35
4.1	Median normal work P85 (hours) by policy across capacity scenarios.	52
4.2	Median urgent hit rate (%) by policy across capacity scenarios.	53

List of Tables

3.1	Lead Time distribution for completed normal and urgent issues.	32
3.2	Arrival summary.	33
3.3	Service time summary: baseline and Trim995.	34
3.4	Parameters for the simulation.	40
4.1	Baseline: normal work predictability.	42
4.2	Baseline: run-level variability of normal work.	43
4.3	Baseline: urgent work performance.	44
4.4	Baseline: run-level variability of urgent work.	45
4.5	Baseline: completion and run-level summary.	45
4.6	Baseline: run-level variability of completion and preemption.	46
4.7	800-server: run-level variability of normal work.	47
4.8	800-server: run-level variability of urgent work.	48
4.9	800-server: run-level variability of finish time and urgent items.	49
4.10	1,200-server: run-level variability of normal work.	49
4.11	1,200-server: run-level variability of urgent work.	50
4.12	1,200-server: run-level variability of finish time and urgent items completed.	51
4.13	Comparison of median run-level metrics: baseline and trimmed.	54
4.14	Comparison of median normal work and urgent work metrics: baseline and trimmed.	55

List of Listings

A.1	README.txt – setup and usage	71
A.2	run_repro.sh – entry script	72
A.3	repro_pipeline.R – data preparation pipeline	73
A.4	validate.R – validation and summary output	78
A.5	Simulation model documentation (AnyLogic export)	84

1 Introduction

Kanban is commonly used in software development and IT service environments to make work visible, manage flow, and improve predictability of delivery. The method is based on lean ideas, but is applied to knowledge work by visualizing work items, limiting Work in Progress, managing flow, and defining policies and processes clearly (Kanban University, [2022]; Ahmad et al., [2018]). In software engineering, this is relevant because work often arrives continuously, priorities can change, and teams need to react to changing requirements and operational problems (Ahmad et al., [2018]; Ozkan et al., [2022]). A main goal of Kanban is predictable flow. Predictability does not mean that every work item finishes at the same time. Rather, it means that teams can make reliable delivery forecasts with an acceptable degree of uncertainty ('Open Guide to Kanban', [2025]). This is why flow-based analysis usually focuses on distributions, percentiles, throughput, and Work in Progress instead of relying on averages alone (Vacanti, [2025]). Studies also show that Kanban can be evaluated using historical flow data. For example, Sjøberg et al. ([2012]) report a case study in which the use of Kanban was associated with shorter Lead Times compared to the earlier Scrum-based setup, while also highlighting that such results must be interpreted carefully because they come from a specific organizational context.

In practice, predictable flow becomes harder to maintain when urgent work enters the system. Urgent items, such as incidents, blockers, or escalated requests, often require faster handling than normal work. In Jira-based workflows, this creates specific operational decisions. Should urgent items simply move ahead in the queue, should they interrupt ongoing work, should part of the available capacity be reserved for them, or should their number be limited through a clearly defined expedite policy? Each of these decisions can improve responsiveness for urgent work, but it can also change the Lead Time distribution of normal work. The problem is not only how urgent work can be completed quickly, but also how much predictability normal work loses when urgent work receives special treatment. Existing research provides useful foundations for this problem, but it does not fully answer it. Kanban in software engineering has been studied through case studies, experience reports, and literature reviews, but Ahmad et al. ([2018]) argue that much of the available evidence stays descriptive and that it is difficult to draw strong conclusions about the value of Kanban from the existing literature. Simulation has also been used to study Kanban systems, especially the behavior of WIP limits and flow under controlled conditions (Anderson et al., [2011]). In addition, simulation studies in incident management show that preemption and priority-based dispatching can be analyzed without disturbing real operations (Assunção et al., [2012]). The specific trade-offs between different urgent-work policies in a Kanban-based workflow are still unclear when the simulation inputs are based on real Jira data.

1 Introduction

This thesis addresses this gap by combining Jira data with a discrete-event simulation. The Public Jira Dataset provides issue and workflow data from public Jira repositories and makes it possible to analyze real software project histories (Montgomery et al., 2025). The dataset and its construction are described in Montgomery et al. (2022). Based on the SPARK project, this thesis derives arrival patterns, an urgent-work threshold, and class-specific service-time proxies from the 2015–2018 window. These inputs are then used in a simulation model that compares four urgent-work policies under the same conditions: Priority Jumping, Preemption, Reserved Capacity, and Strict Expedite WIP Limit. The model represents a Kanban system with two work classes, normal and urgent, and the policies differ in how capacity is shared between them.

The purpose of this thesis is to evaluate how these four urgent-work policies affect the predictability of normal work and urgent-work performance. The contribution is not that simulation is new in this field. Rather, the contribution lies in combining a controlled simulation setup with inputs derived from real Jira data, so that the policy comparison is reproducible and connected to an observed software workflow. This allows the thesis to measure trade-offs that are often discussed in practical Kanban settings, but rarely measured in a directly comparable way. In practice, teams have to choose one of these policies when urgent work appears, but they often have little evidence to base this choice on. A policy that is chosen only to handle urgent work faster can also reduce the predictability of delivery that Kanban is used to improve. By comparing the four policies under the same inputs, this thesis makes these trade-offs visible, both for teams that have to make this decision and for research that aims to study it in a measurable way. The remainder of this thesis is organized as follows. Section 2 presents the theoretical background on predictability, Kanban, flow metrics, and the queueing logic behind the policies. Section 3 describes the preparation of the SPARK Jira data and the simulation model. Section 4 reports the simulation results, and Section 5 discusses the policy trade-offs and limitations of the study.

1.1 Research questions

The research gap and purpose described above lead to four research questions. The first one looks at the Jira data, the second and third compare the four urgent-work policies in the simulation, and the fourth checks whether the results stay stable under different conditions. Together, the four questions cover both the data part and the simulation part of the thesis.

RQ1: How does the Lead Time distribution of normal work differ from that of urgent work in a Kanban system based on real Jira data?

This question establishes the basis for the thesis. It compares normal and urgent work in the SPARK 2015–2018 window and analyzes whether both classes show different Lead Times in the observed Jira data.

RQ2: How do different urgent-work policies affect predictability metrics for normal work?

This question focuses on the simulation results for normal work. It compares Priority Jumping, Preemption, Reserved Capacity, and Strict Expedite WIP Limit considering Lead Time percentiles and variability of normal work. The purpose is to identify how each policy changes the predictability of normal work under the same arrival and service-time inputs.

RQ3: What trade-offs occur between urgent-work performance and normal-work predictability under each policy?

This question evaluates urgent and normal work together. Urgent-work performance is measured through urgent Lead Time percentiles and the share of urgent items completed within the urgent-work threshold. Normal-work predictability is measured through Lead Time percentiles and spread. The question makes it clear whether a policy improves urgent responsiveness at the cost of normal-work predictability, or whether it offers a more balanced outcome.

RQ4: Under which capacity and service-time conditions does each policy perform best?

This question focuses on the robustness of the policy comparison. It analyzes whether the performance of the four policies changes under different capacity levels and under the service-time sensitivity setting. This helps to analyze if the baseline findings depend on the configuration of one specific parameter or remain stable across other simulation conditions.

2 Theoretical Background

Software development has become one of the main value-creation processes of modern organizations over the past decades. At the same time, it is fundamentally different from industrial production processes, from which many of today's management methods are coming from. Ozkan et al. (2022) describe the structural characteristic of software development to be an evolutionary one. Requirements, constraints, and stakeholders change continuously during the course of a project. These changes are driven by changing market needs and evolving technologies. The result of this continuous change in requirements and constraints was the emergence of agile methodologies. The benefit of these methodologies is that they prioritize shorter development cycles and greater adaptability. Ahmad et al. (2018) add that software development has always had recurring problems, including limited reliability, weak responsiveness to change, restricted agility, and high costs. In this context, Kanban was established to address these challenges more effectively.

The roots of this approach lie in manufacturing. Petersen and Wohlin (2011) show how principles of lean manufacturing can be made useful for agile software development. Lean manufacturing focuses on the elimination of waste and the analysis of material flow, which results in improved Lead Times and responsiveness to customers. When these principles are transferred to software development, it becomes clear that classical manufacturing metrics cannot be simply adopted without any changes. The concept of capacity utilization works well in production because it is possible to reliably estimate how many items a machine can produce in a given time unit. For software developers as knowledge workers, this is different, because their behavior is less standardized and productivity varies across individuals and situations. A similar point applies to delivery dates. When software is delivered in small steps over time, the central issue is often not hitting a single fixed deadline but being able to continuously deliver the most important requirements. These characteristics can be described through the concept of *knowledge work*. The *Open Guide to Kanban* defines knowledge work as the creation, application, or management of information through cognitive processes ('Open Guide to Kanban', 2025). Its purpose is to solve complex problems, make decisions, or drive innovation. Such work typically requires expertise, judgment, and collaboration. A main characteristic of knowledge work is its limited visibility. The work itself and the waste associated with it are often not observable to outsiders. In addition, knowledge work involves much greater variation in the size of the work item than industrial production processes, which makes classical concepts of capacity planning even more difficult to apply ('Open Guide to Kanban', 2025). Kanban University (2022) also emphasizes that uncertainty is a natural part of knowledge work. This uncertainty affects how delivery times and delivery expectations can be handled.

2 Theoretical Background

This complexity becomes especially visible in software service operations. Westin (2016) describes how the ICT industry is moving from a product-centered model to a service-centered one. As a result, the requirements for work organization also change. In the ongoing operation of a software service, the focus shifts to maintaining high service quality and meeting service level agreements. At the same time, advance planning becomes more difficult because issues can arise on a daily basis that require immediate action. The amount of work varies greatly, from sporadic maintenance tasks to periods of full team utilization. The size of work also varies strongly, from small corrections to large and complex change requests (Westin, 2016). Ahmad et al. (2018) summarize this by noting that Kanban is applied in an environment that is complex, strongly contextual, and socially embedded, and therefore fundamentally different from the manufacturing conditions from which the method originally emerged. These conditions form the starting point for the following sections. Only against this background does it become clear why flow-based approaches such as Kanban are relevant for IT delivery. Before turning to Kanban as a methodological framework, it is necessary to establish what a well-functioning delivery process should achieve. Predictability, which is the ability to reliably forecast when work will be completed, serves as the central quality dimension of this thesis. Understanding it requires a probabilistic rather than an average-based perspective. Kanban is then introduced as the framework through which predictability can be achieved. Flow metrics provide the measurement instruments that make system behavior observable and steerable. Only once this foundation is in place it becomes possible to identify what threatens it. Urgent work is a recurring management challenge that introduces variability into normally stable systems. How such work is handled, and whether it is handled explicitly at all, is where classes of service become relevant as a response.

2.1 Predictability

In the context of this thesis, *predictability* not only describes a general planning task but also a characteristic of an operationalization system. Vacanti (2025) captures this more briefly by defining a predictable process as one that behaves as expected. Whether a process is considered predictable is therefore judged by the accuracy with which delivery expectations can be set and met. The ‘Open Guide to Kanban’ (2025) describes predictability as one of the three main quality dimensions of a workflow, along with effectiveness and efficiency. It defines predictability as the ability to forecast value delivery within an acceptable degree of uncertainty. Vacanti (2025) clarifies this idea in practical terms by distinguishing between two closely related dimensions. First, a process must behave in a stable and expected manner. Second, based on this behavior, it must be possible to make reliable forecasts about future delivery times. Both dimensions depend on each other. A process that does not behave stable cannot produce reliable forecasts, while forecasts without a basis in observable system behavior remain without practical value.

2.1.1 Variability and averages

The main barrier of achieving predictability in software development is the inherent variability of the development process itself. Petersen and Wohlin (2011) show in their case study in Ericsson that the manufacturing concept of capacity utilization cannot be simply transferred to software development. The reason is that developers, as knowledge workers, do not behave like machines in the production industry. Their productivity can vary substantially across individuals and situations. Petersen and Wohlin (2011) argue that a completely predictable environment would exist only if all observed data points were exactly on the prediction line. This represents an ideal state that cannot be achieved in real development environments. High variance in the flow of requirements makes the planning of development processes more difficult. Vacanti (2025) describes the same problem using statistical language. Cycle Time, Throughput, Work in Progress (WIP), and Work Item Age are not fixed values. They are random variables characterized by probability distributions. As soon as uncertainty and risk shape the process, this becomes visible in all central flow metrics. The practical consequence is that planning on the basis of average values alone can lead to misjudgments. Instead, a probabilistic understanding is required.

This error can also be referred to as the *flaw of the averages*. The idea is that plans based on averages fail on average. This can be explained with a simple example. If ten people each have an on-time probability of 50%, a meeting does not start on time with a probability of 50%, but only with a probability of 0.1%, because all ten participants must be present on time for the meeting to start. When applied to flow metrics, this means that saying a work item takes ten days on average is of limited use for making a delivery commitment because it fails to take variability and delay risk into account. This relationship can also be framed through the assumptions of Little's Law. A process that consistently violates these assumptions, for example, by allowing WIP to grow uncontrolled or by letting items age unnecessarily, is by definition unpredictable. Therefore, variability is not just a minor statistical issue, it indicates a deeper instability within the system (Vacanti, 2025). Variability is not only a theoretical issue. It has also been observed in empirical Kanban studies. In the case study by Sjøberg et al. (2012), the shift from Scrum to Kanban was followed by lower average Lead Times and lower Lead Time variation. For bugs, the standard deviation decreased from 17 to 5 days. For items in product backlog, it decreased from 20 to 9 days. Westin (2016) also refers to data according to which, over an observation period of one year, Lead Time variance decreased by 47%. This suggests that Kanban can not only shorten Lead Times, but also make delivery times more stable. For this thesis, this is relevant because predictability is understood as the stability of Lead Time outcomes, not only as faster delivery.

2.1.2 Measuring and steering

If processes are variable in nature, predictability needs measurement instruments that make this variability visible rather than hiding it behind averages. Westin (2016) summarizes that Lead Time and Due Date Performance are the metrics that show most clearly how predictable a Kanban system is. Lead Time describes how long an item takes to move through the process. Due Date Performance measures what share of items is completed within the promised time frame. This can be assessed through fixed delivery dates or through a comparison between predicted and actual Lead Times (Westin, 2016). Kanban University (2022) adds that the Lead Time distribution can itself be used as an indicator of the predictability of the overall system. A wide distribution of Lead Times indicates greater variation and, as a result, usually lower predictability. In contrast, a tighter distribution points to more stable system behavior. The operationalization of this distribution through percentile lines and the Service Level Expectation is discussed in Section 2.3.

Ahmad et al. (2018) document in their analysis of 23 experience reports that improved delivery predictability is one of the main reported effects of Kanban. At the same time, task switching and unpredictable flow are identified as challenges. The *Open Guide to Kanban* explains the mechanism behind it. Reprioritization of already started work causes items to remain in the system without progress, leading to longer and less predictable Cycle Times (‘Open Guide to Kanban’, 2025). Vacanti (2025) highlights the avoidable aging of work items as the primary factor that undermines the predictability of the process. When items remain in the system longer than their expected duration, either due to blockers, reprioritization, or capacity displacement, an aging effect occurs. This aging effect widens the Cycle Time distribution and reduces the reliability of delivery forecasts. Urgent work, as discussed in Section 2.4, is one of the primary mechanisms through which this aging is introduced into normally stable systems and therefore is a threat to predictability. These sections provide the concepts needed for the analysis and explain why urgent work can threaten predictability. The policies examined in this thesis try to manage this problem.

2.2 Kanban

Kanban describes a strategy for optimizing the flow of value through a system (‘Open Guide to Kanban’, 2025). The term originates from Japan and is commonly translated as “signboard” or “signal card”. In the Toyota Production System, Kanban was used as a signaling mechanism to control production. Restocking of parts was only done when there was downstream demand (Ahmad et al., 2018). This pull principle, which means that work is triggered by actual demand, remains a core foundation of the Kanban approach in knowledge work today (‘Open Guide to Kanban’, 2025).

2 Theoretical Background

The use of Kanban in software development is closely associated with David Anderson, who first applied the approach in 2004 in a software development team at Microsoft in Hyderabad (Ahmad et al., 2018). Since then, Kanban has become widely used in the IT sector. The use of Kanban among observed software teams increased from 39% in 2015 to 50% in 2016. In their systematic mapping study covering the period 2006 to 2016, Ahmad et al. (2018) identified 382 publications, of which 23 were classified as relevant primary studies. This increase in publications shows that Kanban received more academic attention during this period, although research on Kanban was still limited compared to other agile methods. Ozkan et al. (2022) report a similar pattern in their systematic review of the literature. Their findings suggest that organizations seem to prefer Kanban over Scrum, especially when flexibility is a central requirement. They also prefer Kanban when fast reactions to changing priorities and continuous delivery capability are important. In the studies included in their review, many teams lean more toward flow than toward sprint logic.

2.2.1 Kanban as a queueing system

In the theoretical foundation of this thesis, Kanban is understood as a queueing system (Vacanti, 2025). Queueing theory is formally defined as the study of queueing behavior in networks and systems. It provides a mathematical basis for reasoning about mean delay, delay variability, scheduling policies, and capacity planning. Work enters the system as demand, moves through a defined set of states, and leaves it as a complete result. Each of these states can be interpreted as part of a queueing system in which work items are either waiting or being worked on. In queueing theory, the time that a work item is worked on is referred to as its service time, different from its waiting time in the queue (Harchol-Balter, 2013). The total time an item spends in the system, from arrival to completion, combines both. This distinction is important for later analysis because Jira data include workflow-state durations. These durations allow a service-time proxy to be derived separately from waiting time. This queueing perspective is relevant for this thesis, as the policies compared differ in how they allocate capacity between urgent and normal work within a shared queue.

The main control variable of such a system is WIP, which means the amount of work that has been started, but not yet completed. Vacanti (2025) emphasizes that WIP is the main flow metric of a system and that other performance measures, especially Cycle Time and Throughput, are influenced by it. If WIP increases, the average time each item spends in the system also tends to increase. When WIP is kept stable and controlled, queues become less prone to uncontrolled growth, and Cycle Time variation can decline. This relationship is not only an observation, but can also be explained from a queueing perspective, as discussed in more detail in Section 2.3. This logic motivates the pull principle, which applies these ideas at the operational level. New work is pulled into a workflow step only when capacity is available (‘Open Guide to Kanban’, 2025). Unlike push systems, where work may be assigned without considering available capacity, the pull principle helps limit WIP and makes overload visible earlier.

2.2.2 The six practices

Kanban University (2022) describes six general practices that together shape a Kanban system. The first practice, visualize the workflow, makes the state of work items and the overall structure of the workflow visible. This usually happens through a physical or digital Kanban board whose columns represent the different workflow states. Visibility is not only a matter of representation, but a prerequisite for control. Ahmad et al. (2018) identify improved visibility and transparency as the most frequently reported advantage of Kanban in software development. In 16 of the 23 primary studies included in their review, this benefit is explicitly mentioned. Figure 2.1 shows a simplified Kanban board. It illustrates the basic board logic used in Kanban: work items move from left to right, active work is limited through WIP limits (3), and urgent work can be made visible through a separate *expedite lane* (Westin, 2016).

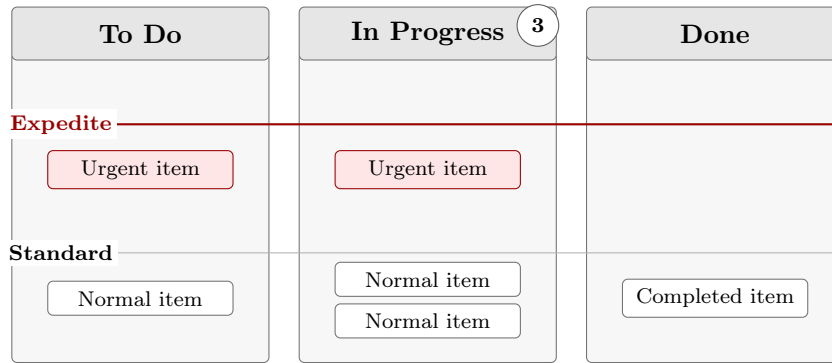


Figure 2.1: Simplified Kanban board. Own illustration, based on Westin (2016).

The second practice, limit Work in Progress, defines an upper bound for the amount of work that may be started at the same time in a specific workflow step or in the system as a whole. According to Kanban University (2022), explicit WIP limits are a central feature of a Kanban system. Such limits force teams to complete the work already started before taking on new work. This makes bottlenecks and overload visible at an early stage.

The third practice, manage flow, shifts the focus from individual work items to the overall movement of work through the system. This includes observing blockers, measuring flow times, and reacting to patterns that indicate instability or bottlenecks. Petersen and Wohlin (2011) illustrate this in an industrial case study at Ericsson AB, where a cumulative flow diagram was used to visualize bottlenecks and variation in flow.

The fourth practice, make policies explicit, requires that the rules of the system are documented and understood by all participants. These rules include, for example, selection

2 Theoretical Background

criteria for work items, definitions of done, WIP limits, and priority rules (Kanban University, 2022). Explicit policies create a shared basis for making decisions and make process changes easier to handle. In the absence of explicit policies, teams tend to handle urgent work through informal ad hoc decisions that are neither consistent nor observable (‘Open Guide to Kanban’, 2025). The four policies compared in this study (Priority Jumping, Preemption, Reserved Capacity, and Strict Expedite WIP Limit) can be understood as different attempts to make exactly these implicit decisions explicit and thereby controllable.

The fifth practice, implement feedback loops, refers to regular feedback formats at different levels, such as team meetings at the board, replenishment meetings, and periodic retrospectives (Kanban University, 2022). These loops connect observed flow metrics with concrete process adaptations.

The sixth practice, improve collaboratively, focuses on the evolutionary character of Kanban. Change is introduced step by step, based on data and collective learning, rather than through a one-time top-down redesign (Kanban University, 2022). Ahmad et al. (2018) also note that Kanban can be introduced into existing processes. This helps explain its broad practical use.

2.2.3 Empirical evidence

The literature generally reports positive effects of Kanban, but the methodological quality of the evidence varies across studies. Sjøberg et al. (2012) present a quantitative case study at Software Innovation in Norway based on more than 12,000 work items and show that average Lead Times decreased after the introduction of Kanban. For bugs, average Lead Time fell from 12 to 5 days, and for product backlog items from 14 to 7 days. Particularly relevant for this thesis is that not only the average, but also the variation in Lead Time decreased. The standard deviation fell from 17 to 5 days for bugs and from 20 to 9 days for product backlog items. This points to improved delivery predictability. In addition, the authors report a 10% reduction in weighted bugs, a 26% reduction in critical blocking bugs, and a 21% increase in churn-based productivity (lines added, deleted, or modified) for product backlog items. Anderson et al. (2011) use a discrete-event simulation to show that WIP limits can also be represented in the cumulative flow diagram. Systems with explicit WIP limits show a more orderly pattern, whereas systems without such limits show more irregular flow behavior. This supports the theoretical link between WIP control and flow stability.

A mapping study of Kanban in software development identifies 15 reported advantages. In addition to improved visibility, transparency, and workflow management, higher predictability is explicitly mentioned in 6 of the 23 experience reports reviewed. At the same time, the authors emphasize that much of the available evidence comes from experience reports rather than systematic empirical studies. Only a limited number of evaluations are methodologically robust (Ahmad et al., 2018). This methodological gap motivates

the simulation based approach in the present thesis, which aims to provide empirically grounded evidence on a specific aspect of Kanban.

2.2.4 Kanban in the service context

The relevance of Kanban becomes visible in service contexts. In an action research study conducted over four cycles in a Finnish ICT company, Westin (2016) argues that Scrum reaches structural limitations in service operations. Urgent issues that arise outside the current iteration are difficult to integrate without either destabilizing the sprint content or delaying urgent work. Kanban offers an advantage in this context because it does not rely on fixed iteration boundaries and is better suited to environments with continuous incoming work, interruptions, and changing priorities. Westin (2016) also points out the possibility of differentiating WIP limits by work item type or class of service. This allows different types of demand to be handled in a controlled way, making Kanban relevant for environments in which planned and unplanned work must be processed in parallel. Ozkan et al. (2022) also report in their systematic literature review that organizations often prefer a shift from Scrum to Kanban or to hybrid forms when flexibility is limited, priorities change frequently, and the current workload lacks transparency. Teams operating in dynamic environments benefit from the strong flow orientation of the Kanban approach.

In general, the literature suggests that Kanban provides a suitable framework for IT operations and software development. This is especially the case when demand varies and urgent work must be handled alongside regular work. At the same time, the findings indicate that the effects of Kanban do not result from visualization alone, but from the consistent use of WIP control, the pull principle, and explicit policies. As the following section shows, the ability to make reliable predictability statements is closely linked to how flow is measured and evaluated.

2.3 Flow metrics

As a flow-based system, Kanban generates continuous data on where work items are in the workflow and how they move through it. These data become useful only when they are translated into metrics that quantify delivery speed and help diagnose stability problems and bottlenecks. Flow metrics are therefore a prerequisite to make system performance observable and to decide whether delivery promises are realistic and can be made reliably (Vacanti, 2025). This section introduces the central flow metrics, explains the mathematical relationships between them, and presents the most important visualizations for practical evaluation. In this way, it establishes the analytical foundation for the research questions of this thesis.

2.3.1 The four central flow metrics

Vacanti (2025) describes four core metrics that together provide the foundation of system behavior: *Work in Progress*, *Cycle Time*, *Throughput*, and *Work Item Age*. This thesis treats these terms as defined technical concepts and capitalizes them accordingly throughout.

Work in Progress refers to the amount of work that has already been started but has not yet been completed within an observed part of the system. Work in Progress is not only a countable quantity, but the central flow metric of the system. As already discussed in Section 2.2, WIP influences the other performance measures. Vacanti (2025) argues that by controlling WIP, a team can indirectly influence observable Cycle Time behavior without having to change individual process steps.

Cycle Time describes the time it takes for a work item to move through a defined part of the workflow (Vacanti, 2025). The *Open Guide to Kanban* uses the related term elapsed time since first touch, or elapsed time from started to finished (ETSF), and emphasizes the period from a defined starting point until completion ('Open Guide to Kanban', 2025). For analytical purposes, it is important to define this starting point clearly. In the theoretical discussion of this thesis, the term Cycle Time follows Vacanti's framework and means the time from the commitment point to completion (Vacanti, 2025). For the analysis, the term *Lead Time* is used to describe the time from ticket creation to completion based on Jira data. This corresponds to *Customer Lead Time* as defined in the *Official Kanban Guide* (Kanban University, 2022). Both concepts measure the duration of delivery and are used in this thesis to assess predictability. The distinction is noted for terminological clarity. Regardless of the exact definition, both metrics mainly describe how fast work is delivered.

Throughput measures how many work items a system completes within a defined time frame, for example per week (Vacanti, 2025). Unlike planning-based estimates or size assumptions, Throughput is based on actual observed completions. The *Open Guide to Kanban* also refers to this concept as *Delivery Rate* ('Open Guide to Kanban', 2025). Throughput is especially relevant for statements about delivery capacity at the system level.

Work Item Age measures how much time a work item, that has not yet been completed, has already spent in the system ('Open Guide to Kanban', 2025). In contrast to Cycle Time, this metric is already available while the item is still being processed, and therefore has an operational early warning function. If the age of an item exceeds its historically typical range, this can indicate blockers, waiting states, or other systematic problems. Vacanti (2025) describes in this context the so called *aging effect*. As the time already spent in the system increases, the probability also increases that the item will remain in the system for longer.

2.3.2 Little's Law

The relationship between the central flow metrics can be described by Little's Law. In its well-known form, it states:

$$L = \lambda \times W$$

Here, L represents the average number of work items in the system, λ the average arrival rate into the system, and W the average time an item spends in the system (Vacanti, 2025). Translated into the Kanban context, this can be expressed as:

$$\text{Average Cycle Time} = \frac{\text{Average WIP}}{\text{Average Throughput}}$$

For Kanban systems, this relationship is central because it shows that Cycle Time, WIP, and Throughput cannot be considered independently of each other. If Throughput stays constant, a higher WIP leads to longer average times in the system. Conversely, a reduction in WIP can shorten average Cycle Time without increasing Throughput. At the same time, Vacanti (2025) explains that Little's Law is only useful under specific conditions, especially when the underlying system assumptions are met. In practice, the law is understood less as an exact control formula and more as an analytical reference frame. If the expected Cycle Time derived from WIP and Throughput differs substantially from the observed values, this may indicate instability, blockers, disruptions, or problems in how the system is defined or measured. Vacanti (2025) further argues that Little's Law is important not mainly because of the calculation itself but because of the assumptions behind it. If a process systematically violates these assumptions, for example, by allowing WIP to grow unchecked or by letting items age unnecessarily, it becomes less stable and harder to predict. Understanding which policies cause such violations is central to the research questions of this thesis.

2.3.3 Averages

A central argument made by Vacanti (2025) is that average values for Cycle Time and Throughput often provide an insufficient picture in practice. The reason lies in the typical variation of these metrics. In real software and service systems, Cycle Times are usually not symmetrically distributed but are right-skewed. Many work items are completed relatively quickly, while a smaller number remain in the system for a very long time. These long durations pull the average upward without adequately reflecting the actual spread of the distribution. An average value of ten days therefore does not mean that

2 Theoretical Background

most work items take around ten days. The median may lie considerably below that value, while a relevant share of items remains in the system much longer (Vacanti, 2025). If communication about delivery times is based only on averages, there is a risk of creating systematically optimistic expectations.

For this reason, Vacanti (2025) recommends working with percentiles. The $P50$ value, which is also the median, marks the point below which 50% of all work items are completed. The $P85$ value marks the point below which 85% of all work items are completed. For delivery promises, higher percentiles such as $P85$ are often more useful because they provide a more robust and more cautious reference than the average. Figure 2.2 illustrates this logic. The percentile lines visualize how *Cycle Times* are distributed across completed work items and why a value such as $P85$ provides a more suitable basis for delivery expectations than an average.

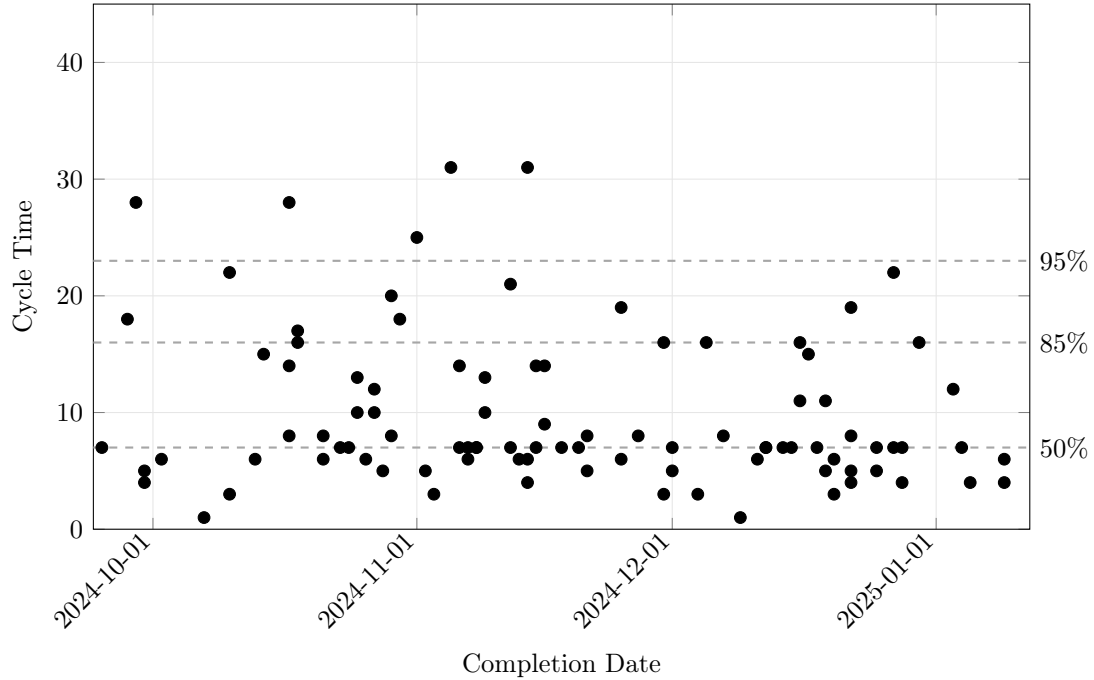


Figure 2.2: Cycle time scatterplot with percentile lines. Own illustration, based on Vacanti (2025).

Percentile lines also provide information all together, not only individually. The distance between them reflects the stability of the system. If $P50$ and $P95$ lie close to each other, most work items complete within a similar time frame, and the process can be considered predictable. If $P95$ lies far above $P50$, the typical item finishes quickly, but a relevant share of items remains in the system much longer. In such cases, predictability is reduced even when the median itself seems to be acceptable. Vacanti (2025)

2 Theoretical Background

illustrates this through an example in which P50 is 20 days, P85 is 54 days, and P95 is 75 days. Moving from 70% to 85% confidence in this distribution more than doubles the resulting delivery promise. A wider spread between percentile lines signals greater variability and lower predictability (Vacanti, 2025). In this thesis, the distance between P95 and P50 is used as the spread measure for normal-work Lead Times. It captures how far the tail of the distribution lies above the typical case. The *Open Guide to Kanban* follows the same logic and recommends probabilistic statements such as “85% of work items will be finished in eight days or less” (‘Open Guide to Kanban’, 2025). The advantage of such statements is that they do not ignore uncertainty but make it visible.

2.3.4 Service Level Expectation (SLE)

The concept of *Service Level Expectation* (SLE) follows directly from the percentile-based logic. An SLE is an empirically derived statement about the time frame within which a work item is expected to be completed with a given probability (Vacanti, 2025). The *Open Guide to Kanban* defines the SLE as a forecast of how long a work item should take from a defined starting point to a defined endpoint (‘Open Guide to Kanban’, 2025).

A Service Level Expectation is based on historical Cycle Time data or historical elapsed time from started to finished data. If sufficient historical data are not available, it may also be based on an estimate. At the same time, the guide notes that past observations do not guarantee future system behavior (‘Open Guide to Kanban’, 2025). This point is relevant in the context of system changes, such as the introduction of new policies. From the SLE follows an observable performance metric, which is the share of items that meet the expected time frame. In service and incident-management contexts, similar measures are often discussed as SLA compliance, because they compare actual completion times with an agreed service target (Abdulahi, 2023). For this thesis, the SLE is the primary conceptual reference of predictability. A shift in the SLE of normal-work items under different expedite policies is the main observable indicator for answering RQ1, RQ2, and RQ3. This change appears as a broader spread in the Cycle Time distribution or an increase in the higher percentiles. The related simulation metric is called the urgent hit rate. It measures the fraction of urgent items completed within the derived urgent-work threshold. This wording is used because the threshold is not a contractual SLA, but a benchmark derived from the observed urgent Lead Time distribution.

2.3.5 Scope and relevance for this thesis

Beyond the four core metrics, the *Open Guide to Kanban* also mentions additional metrics for analytical purposes (‘Open Guide to Kanban’, 2025). These include *Flow Efficiency*, which describes the ratio of active processing time to the total time a work item spends in the system. Low *Flow Efficiency* may indicate that a share of total elapsed time consists of waiting, idle time, or blocked time rather than active work. The guide also refers to

2 Theoretical Background

Blocked Elapsed Time for Finished Items (BETFI) as a metric to analyze the blocked fraction of completed work items (‘Open Guide to Kanban’, 2025). For this thesis, these metrics are not directly relevant. The main focus lies on how expedite policies affect predictability, variation, and delivery stability.

The metrics introduced in this section form the analytical foundation for the analysis. The question of whether and how different expedite policies affect the predictability of normal-work items is operationalized through Lead Time percentiles and through the spread between P95 and P50, which captures the width of the upper part of the distribution. The Service Level Expectation serves as the main reference frame for predictability because it provides a probabilistic statement about the reliability with which normal-work items can be completed within an expected time frame. For urgent work, performance is operationalized through urgent Lead Time percentiles and the urgent hit rate, measured against the urgent-work threshold derived from the observed urgent Lead Time distribution.

2.4 Urgent work

In software development and IT service processes, some work needs to be prioritized over normal work for operational, economic, or contractual reasons (Li et al., 2024). In practice, this type of work appears under different labels, such as expedites, high-severity tickets, blocker issues, or more generally urgent work. What these categories have in common is that they intervene in the regular flow of work. They change priorities, shift planned sequences, and bind capacity that was originally intended for other work items (Vacanti, 2025). Urgent work is therefore not an exception, but a recurring issue of real development and service processes (Assunção et al., 2012). The findings discussed in the following show that measurable consequences are associated with it at different levels. For the productivity of individual developers, for the compliance with service level agreements, and for the predictability of the entire system (Shakeri Hossein Abad et al., 2018). This section approaches the topic from four perspectives. The empirical frequency of priority changes, the cognitive and operational costs of interruptions, the effects of different scheduling policies under different load conditions, and the resulting system-level effects on flow. Together, these perspectives establish that urgent work creates costs at every level of analysis. This affects not only individual developers, but also the system as a whole. The policy choice for handling urgent work has measurable consequences for predictability.

2.4.1 Frequency and patterns of priority changes

Li et al. (2024) show that urgent work is not a rare exception in software practice in an empirical study of 32 Apache open-source projects. Based on 223,355 bugs from Jira data, the authors find that 8.3% of all bugs experienced at least one priority change. In

2 Theoretical Background

organizations with thousands of active issues, this implies that a relevant number of work items are regularly affected by later priority intervention. Such interruptions can change how work flows through the process. Focusing on the timing of these changes within the lifecycle of a bug, Li et al. (2024) show when priority adjustments are made. They find that almost half of all priority changes occur before work on the respective bug has even started. A considerable share of these changes affects sequencing decisions before processing even begins. At the same time, the authors show that most affected bugs are reprioritized only once. 87.9% of bugs with a priority change experience exactly one such change (Li et al., 2024).

Li et al. (2024) also report that bugs with higher initial priorities are more likely to be reprioritized later. Across the projects studied, Blocker and Critical bugs show a higher probability of priority change than lower priority bugs. This suggests that the higher the initial priority of a bug, the greater the probability that its priority will later be changed. For the analysis in this thesis, urgent work is operationalized using exactly these priority labels. Tickets classified as Blocker or Critical in Jira are treated as urgent, while all remaining priorities are treated as normal work. This operationalization is consistent with the findings of Li et al. (2024) and reflects common handling of priorities in issue tracking systems. In addition, bugs with priority changes are related to higher code complexity and higher communication complexity than bugs whose priority remains unchanged. Among other indicators, this is measured using metrics such as lines of code, the count of modified files and packages, the total number of comments, the average comment length, and the number of individuals who participate in commenting. Urgent work appears not only as a recurring phenomenon, but also as work that is on average more complex and more communication intensive. This makes priority interventions more disruptive because these work items often take longer to process and may require coordination between teams.

2.4.2 Cognitive costs of interruptions

Priority changes are not only a planning problem, but also affect the cognitive work process of developers. Shakeri Hossein Abad et al. (2018) analyze in a mixed method study which factors make task switching and interruptions particularly disruptive in software development. Their study combines a longitudinal analysis of 4,910 recorded tasks of 17 professional software developers over a period of 1.6 years with a survey of 132 developers from companies of different sizes, including Microsoft, Ericsson, Bosch, and Cisco. Regarding the frequency of task switching, Shakeri Hossein Abad et al. (2018) report that developers switch almost 59% of their daily tasks and that around 40% of these switches involve a context change. They also show that 29% of interrupted or switched tasks are never worked on again. In addition, a typical developer day is split into many short work sessions of around 15 to 30 minutes. Another 15 to 30 minutes may be needed after an interruption to reconstruct the previous work context. The reasons developers give for voluntary task switching are relevant because they show how interruption behavior is linked to priority dynamics. Shakeri Hossein Abad et al. (2018)

2 Theoretical Background

identify being blocked in a task, being sidetracked to other tasks, planning issues, priority changes, missing information, and boredom with complex or lengthy work as common triggers. Priority changes appear not only as a planning issue, but also as a recurring trigger of interruption and switching behavior at the individual work level.

For this thesis, the findings on priority change are especially relevant. Shakeri Hossein Abad et al. (2018) show that programming tasks are paused for longer than testing tasks when developers switch to a higher priority task. The difference is statistically significant ($p = 0.002$). Furthermore, 111 of the 132 survey participants, corresponding to 84%, agreed or strongly agreed that immediate and unplanned requests, such as requests for priority change, are disruptive. Programming and testing tasks appear particularly vulnerable in this context because they place high demands on problem solving and working memory. If urgent requests trigger switching, suspend work already in progress, and require context reconstruction before resumed work can continue, interruption costs accumulate. These interruption costs appear as additional delay in the flow of regular work. In a WIP-limited system, these delays do not affect the interrupted developer but widen the Cycle Time distribution of the displaced items, and thereby reduce the predictability of normal work. This dynamic is examined in the following sections from a scheduling and flow perspective.

2.4.3 Preemption and priority policies

At the system level, the question is which dispatching policy is most effective for tickets of different priority, and when interrupting work already in progress is advantageous. Assunção et al. (2012) investigate this with a discrete-event simulator fed with real workload data from a large IT service provider (17,486 tickets over four months). Their workload analysis shows that many tickets miss their SLA deadline not because processing is slow, but because they wait too long for assignment, and that once processing starts, a ticket is usually finished even if higher priority tickets arrive. SLA compliance therefore depends not only on capacity, but also on the sequencing logic of the system. Comparing several dispatching policies, including preemptive variants, they find that preemption with low overhead can improve the handling of high severity tickets, while preemption with full restart worsens low severity tickets. The decisive factor is the overhead threshold. Preemption stops being advantageous once the interruption cost exceeds about 45% of processing time. Since the reported slowdown of interrupted work ranges from 5% to 40%, preemption can often be beneficial under realistic conditions (Assunção et al., 2012). This is directly relevant to the Preemption policy examined here, although its effect in a WIP-limited Kanban system requires further investigation.

Abdulahi (2023) examines how different priority strategies behave under different load conditions, using a discrete-event simulation based on real ticket data from a Finnish telecommunications company. The compared queue disciplines range from First In, First Out to priority- and SLA-based ordering. Although these disciplines do not correspond exactly to the four expedite policies analyzed in this thesis, the study remains relevant

2 Theoretical Background

because it illustrates the general trade-off that becomes visible under load: urgent work can be protected more effectively, but often at the cost of increased delay for lower priority items. Under a balanced load, all disciplines reach similar total SLA compliance, and differences appear only at the level of priority classes. Under overload, the trade-off becomes drastic: FIFO collapses to 3.92% total SLA compliance, while priority-based disciplines reach 57.65% (PBQ) and 58.94% (PBQSLA) by protecting the top priority classes almost completely and pushing the lowest class (P4) down to 4.55% (Abdulahi, 2023). Priority-based policies thus protect critical tickets effectively under load, but systematically shift SLA violations to lower priority work.

2.4.4 Flow debt

From a queueing theory perspective, the cost that urgent work imposes on normal work is not an accidental side effect of poor system design. It is a structural consequence of how priority-based scheduling operates. Harchol-Balter (2013) describes scheduling as a mechanism used not only to reduce mean response time. Scheduling is also used to provide differentiated levels of service where some class of jobs are guaranteed lower mean delay than other classes. This differentiation does not come without cost to the system as a whole. The main difference in priority scheduling is between non-preemptive and preemptive policies. Under a non-preemptive priority policy, a job runs to completion once it starts service, even if a higher priority job arrives in the meantime. Under a preemptive priority policy, the job in service is interrupted as soon as a higher priority job arrives, and the higher priority job begins service immediately (Harchol-Balter, 2013). Both types guarantee that higher priority jobs have lower waiting times than they would under simple FCFS. The mechanism through which this benefit is achieved differs. The same applies to the cost structure that is placed on lower priority work.

Under non-preemptive priority scheduling, a lower priority job must wait behind jobs of the same or higher priority that were already in the queue when it arrived. It must also wait behind all higher priority jobs that arrive while it is waiting (Harchol-Balter, 2013). This means that the waiting time for a lower priority job increases with the volume of higher priority arrivals and the overall load of the system. Under preemptive priority scheduling, higher priority jobs perform even better. A higher priority job is less affected by the variability of lower priority work, because it only has to compete with jobs of the same or higher priority. Conversely, lower priority jobs are affected by the remaining variability in the system, including repeated interruptions (Harchol-Balter, 2013). In both cases, the conclusion is the same, that improving the response time of higher priority work increases the waiting time of lower priority work. The total work in the system remains unchanged. What scheduling policies redistribute is not work, but delay.

The empirical and theoretical perspectives discussed in this section, from the frequency of priority changes, through the cognitive costs of interruptions, to the structural consequences of priority-based scheduling, describe different aspects of the same problem.

2 Theoretical Background

Vacanti (2025) summarizes the system-level side of these effects under the term *flow debt*. Flow debt arises when the Cycle Time of selected prioritized work items is shortened only because other already started items remain in the system for longer. A typical example is the prioritized handling of expedite work. As soon as an expedite item enters the system, it can, depending on the policy, be assigned higher priority than other items and occupy available capacity. If there is no free capacity available, work that is already in progress may be postponed so that the urgent item can be processed faster (Vacanti, 2025). From the perspective of the affected normal-work items, time in the system increases even though the content of that work has not changed. According to Vacanti (2025), this debt is paid back in two main ways. Either the postponed items are eventually completed, but with substantially longer Cycle Times. In that case, historical Cycle Time data loses value as a basis for delivery promises. Or, the delayed items are removed from the system before completion because their value has disappeared in the meantime. In that case, the work already invested is wasted. In both cases, predictability decreases.

Vacanti (2025) further distinguishes, by analogy to financial theory, between three forms of flow debt: hedge, speculative, and Ponzi. In a hedge state, expedites remain exceptions, and the system is still able to handle the delayed work. In a speculative state, there is almost always at least one expedite item in the system, so that WIP limits are under continuous pressure. In a Ponzi state, nearly all work is treated as urgent. Under such conditions, stable normal flow is hardly possible anymore (Vacanti, 2025). This typology shows that the main problem is not a few isolated expedite items, but the repeated accumulation of urgent work. As a countermeasure, it is referred to the principle of slack. Slack means intentionally reserved free capacity. Similar to logistics systems, the aim is to handle temporary load peaks without harming regular deliveries. The more strongly a system deviates from a stable FIFO logic, the more difficult predictability becomes. Slack can help limit this deviation and make it more controllable. Vacanti (2025) also notes that Flow Debt is not exclusive to explicitly designed expedite lanes. Any pull policy that systematically favors certain items over others, whether through formal priority rules or informal ad hoc decisions, can produce the same effect. This point is important for the comparison in this thesis, as all four policies can be understood as different forms of such a deviation.

2.4.5 Implications for this thesis: urgent work

The concepts discussed in this section provide the basis for the research questions of this thesis. Urgent work is not only a theoretical idea, but can also be observed in real work systems. It affects individuals, processes, and the system as a whole. This creates a key tension that urgent work may need to be prioritized to protect critical goals, but this prioritization can also reduce the predictability of normal work. It is precisely within this tension that the policies compared later in this thesis are located. The four policies compared in this thesis (Priority Jumping, Preemption, Reserved Capacity, and Strict Expedite WIP Limit) each represent a different approach to managing this tension. Sec-

tion [2.5](#) investigates the structural rules through which these trade-offs can be organized.

2.5 Classes of service

Classes of service (CoS) are a structural instrument through which Kanban systems respond to the reality described in Section [2.4](#) on urgent work. This reality is that not all work items have the same urgency. If these differences are not made explicit, urgent and normal work may be handled under the same rules, making flow harder to control. This section defines the concept, introduces the four archetypes commonly discussed in practice, examines their effects on predictability with reference to practical experience and simulation results, and finally discusses the conceptual limits of this approach.

2.5.1 Definition and operational core

The *Official Kanban Guide* defines a *class of service* as a specific level of service that is applied to the handling of work items through a defined set of policies. A class of service may apply to one or more *work item types*. Its selection can be based on relative value, risk, or *cost of delay*. Cost of delay describes the rate at which the expected value of a work item sinks over time and therefore captures both urgency and impact. The higher the delay cost, the stronger the justification for the priority treatment. Consequently, the greater the potential impact on the predictability of lower priority work (Kanban University, [2022](#)). Vacanti ([2025](#)) explains the main idea of the concept in more detail. He describes classes of service as a policy, or a set of policies, that manages the sequence in which work items are pulled through a process after they have been committed. In other words, this applies after the point at which they count as Work in Progress. From this, three important implications follow.

First, classes of service are not the same as work item types. A work item type describes what an item is, whereas a class of service regulates the order in which it is processed. Second, classes of service become relevant only after the commitment point. As long as an item remains in the backlog, its later handling within the process is not yet operationally fixed. Even when an item is taken into the system, it may still be foreseeable that its deadline can be met without prioritized treatment. In such a case, a special class of service is not necessary. Third, classes of service are not the same as deciding which items enter the system during replenishment. The selection rules for entry into the system and the sequencing rules within the system are two different decisions (Vacanti, [2025](#)).

2.5.2 Practical implementation on the board

Westin ([2016](#)) describes the conceptual and the practical implications of classes of service in the Kanban context. At a conceptual level, WIP limits can be differentiated through

2 Theoretical Background

swimlanes or colored cards in order to make work item types or classes of service visible. In the Kanban literature, this is commonly linked to categories such as expedite, fixed date, standard, and intangible items. In this way, different categories of work can be visible on the board. At an empirical level, Westin (2016) describes the introduction of a three-level prioritization scheme in a real service team. The category “high” included urgent bug fixes, work with approaching deadlines, and critical system updates. “Normal” referred to standard work, while “low” covered optional tasks that were not time sensitive. These priorities were visualized on the board through swimlanes, with higher priority items at the top and lower priority items at the bottom. In weekly meetings, work items were reviewed systematically from top to bottom and from right to left. According to Westin (2016), the team perceived this visibility of priorities as helpful because it supported a stronger focus on critical work.

At the same time, Westin (2016) also reports problems in the practical implementation. When teams introduced strict WIP limits, they often bypassed them because it seemed more efficient in daily work. For example, maintenance tasks were performed in parallel with service updates because using the same deployment window was perceived as more sensible than strictly following the limit. In addition, the board was not always updated simultaneously with the actual work, so that WIP limits were sometimes adjusted only afterwards. At the end of this cycle, the team concluded that the specific way in which WIP limits had been implemented in this format was counterproductive. This finding shows that classes of service may be harder to apply in practice than the concept itself suggests. Westin (2016) describes a similar problem in the context of environments with multiple customers. He refers to a case in the literature in which urgent requirements from one customer reduced the time available for others, and thereby created prioritization problems. The solution was to define standard priority policies, communicate them transparently to customers, and require customers to coordinate urgent priority changes among themselves. This directly reflects one of the core Kanban principles: making policies explicit.

2.5.3 The four archetypes

The Kanban University (2022) and Vacanti (2025) both refer to four widely used archetypes of classes of service: *expedite*, *fixed-date*, *standard*, and *intangible*. Expedite is used for work items whose delay would have immediate and severe consequences. Fixed-date refers to work items for which the negative consequences of delay happen at a specific point in time. Standard describes work for which the consequences of delay are considered normal under the given circumstances, meaning neither exceptionally high nor tied to a fixed deadline. Intangible applies to work items whose consequences of delay are difficult to determine or initially unclear. At the same time, Vacanti (2025) emphasizes that these four archetypes should not be understood as a closed list. In principle, any policy that influences the sequence in which work is pulled through a process can be understood as a class of service, regardless of whether it is formally labeled as such. Even processes without

2 Theoretical Background

explicit CoS labels often operate in practice with rules that give certain items priority. The question is therefore not whether classes of service exist, but whether they are made explicit and designed consciously. The functioning of the expedite class is explained in the *Official Kanban Guide* through the metaphor of emergency vehicles (Kanban University, 2022). Expedite items can, like vehicles with blue lights, move through the system with priority, while other work has to give way. The consequence is a shorter Lead Time for the prioritized items, while the rest of the work takes on the corresponding load.

2.5.4 Simulation based evidence

Vacanti (2025) quantifies the effects of different classes-of-service policies through a simulation of a WIP-limited process. The simulated workflow has three steps: Specifying with a WIP limit of 2, Development with a WIP limit of 2, and Test with a WIP limit of 1. The backlog contains 50 items, and each item requires exactly 10 days per step. Across all scenarios, the $P85$ value is used as the main measure of expected Cycle Time. In the first scenario, a strict FIFO pull system is simulated without classes of service. The result is a $P85$ Cycle Time of 50 days, with a Cycle Time range between 30 and 50 days. This scenario forms the baseline of a system with low variability and high predictability (see Figure 2.3, Scenario 1). In the second scenario, FIFO is replaced by random pulling, again without classes of service. The $P85$ increases from 50 to 60 days, that is, by 20%, and the Cycle Time distribution widens (Scenario 2). Even this deviation from a strict FIFO principle decreases predictability, despite the absence of explicit priority rules. In the third scenario, the system returns to FIFO for standard items and adds an expedite class of service. At every point in time, exactly one expedite item is present in the system. In the simulation, this item is prioritized over standard items and is allowed to exceed the WIP limits. The $P85$ increases to 65 days (Scenario 3). This finding is noteworthy because even a formally limited expedite class of service reduces predictability more strongly than random pulling without explicit expedite rules, although FIFO is kept for standard work. In the fourth scenario, random pulling for standard items is combined with a permanent expedite class of service. The $P85$ increases to 100 days, and the Cycle Times spread between 40 and 170 days (Scenario 4) (Vacanti, 2025). Compared with the baseline, not only does the distribution become considerably wider, but the maximum Cycle Time also more than triples. These four scenarios define a range of possible predictability outcomes that provides the point of reference for this thesis. The policies under investigation can be positioned along this range. From configurations closer to the stable FIFO baseline to configurations that approach the worst case scenario of permanent expedite presence combined with random pulling.

2 Theoretical Background

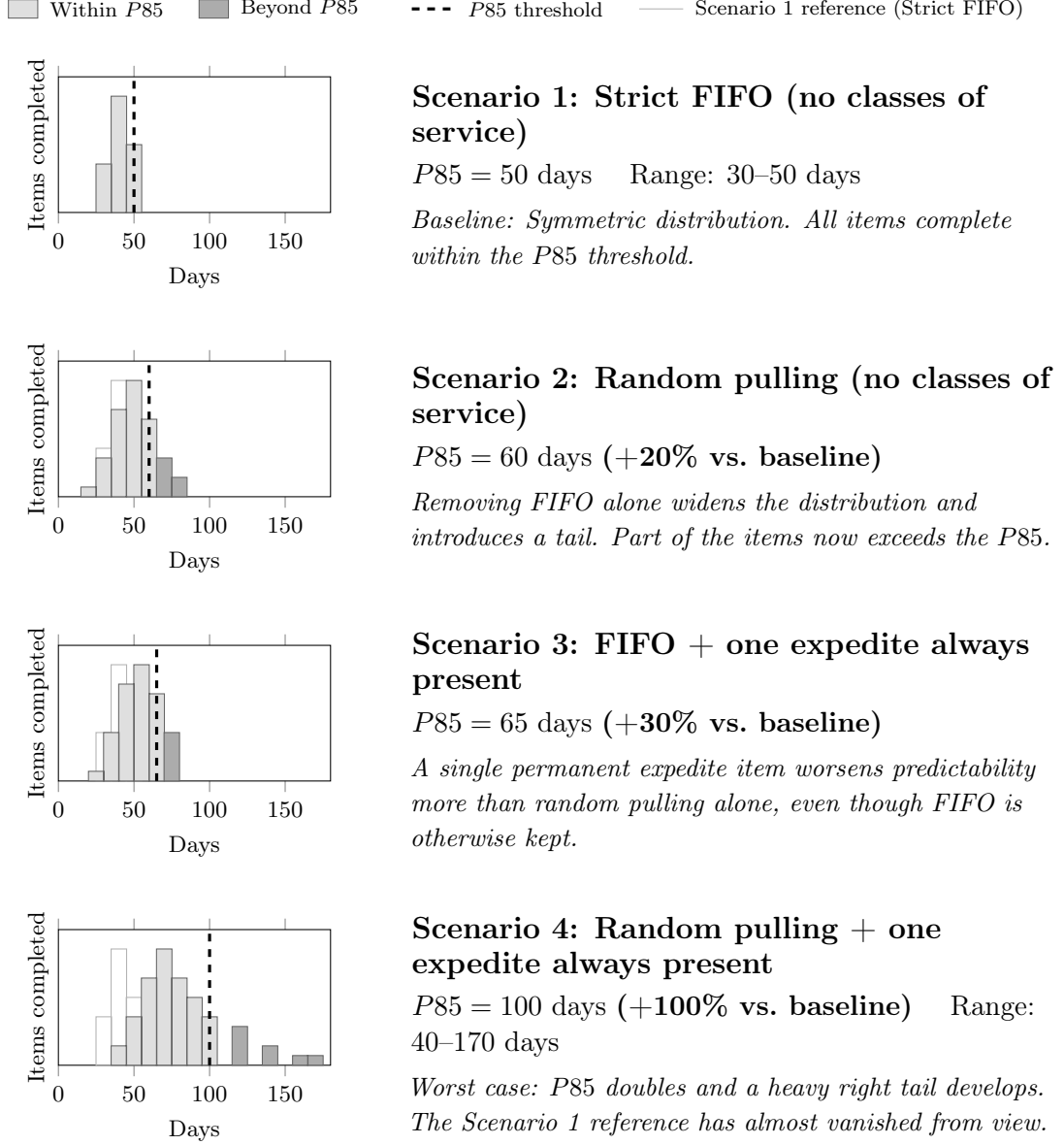


Figure 2.3: Comparison of the four simulation scenarios. Own illustration, based on Vacanti (2025).

2 Theoretical Background

Vacanti (2025) emphasizes that in this experiment all items are of equal size. In real processes, work items often differ in size. This adds more variability and can make the distribution even less favorable in practice. The conclusion is therefore that classes of service can create a deviation from the conditions under which stable and predictable flow patterns come to be. This deviation becomes visible in the form of flow debt and reduced predictability.

A follow-up problem of a functioning expedite lane arises. As soon as it becomes visible within a team that expedite items are completed much faster than standard items, an incentive arises to classify as many work items as possible as expedite. This is not necessarily a sign of bad behavior, but a plausible reaction to observable differences in delivery speed. The effect is predictable, as the expedite WIP limit comes under pressure and becomes increasingly difficult to maintain. What was originally introduced as a strict exception can slowly become the new normal, while the standard class becomes the category for everything that is not treated as urgent (Vacanti, 2025). At the same time, strict FIFO queueing, which represents the ideal scenario in the simulation, is not always feasible in practice. Vacanti (2025) points to structural reasons for this. If a unit of capacity becomes available, but that team member does not have the necessary expertise to work on the oldest waiting item, FIFO cannot be implemented consistently. As an alternative, the First in First Served (FIFS) principle is recommended. Under FIFS, items are usually processed in arrival order, but this rule does not apply in every situation. FIFS is therefore less restrictive than FIFO, while still keeping the basic logic of sequence-based rather than priority-based processing. In practice, it tends to protect predictability better than random pulling or an uncontrolled expedite policy.

The four policies compared in this thesis relate to the expedite archetype in different ways. Priority Jumping and Preemption represent active sequencing interventions that allow expedite items to bypass or displace normal work. Reserved Capacity tackles the issue by assigning a specific portion of capacity to urgent work in advance. In queueing theory terms, it is the only one of the four policies that is not work-conserving. It withholds capacity even when work is waiting in order to reserve availability for urgent arrivals (Harchol-Balter, 2013). Strict Expedite WIP Limit keeps the expedite lane, but limits how many expedite items can be active at the same time. Each of these can be understood as a different operationalization of how the expedite class of service is used.

2.5.5 The four expedite policies

The concepts introduced in the previous sections, classes of service, priority-based scheduling, and flow debt, provide the theoretical background against which the four policies compared in this thesis can be positioned. Each of them represents a different answer to the same operational question: What happens to the system when an urgent item arrives and the capacity is already in use? The following defines each policy in terms of its operating logic, its relationship to the preemptive and non-preemptive distinction

2 Theoretical Background

introduced in Section 2.4 and the cost structure it implies for normal work.

Priority Jumping is a non-preemptive policy. When an urgent item arrives, it is moved to the front of the queue and given priority over normal items that are waiting to be started. Work that is already in progress continues until completion. Vacanti (2025) describes this as the standard operating mode of most expedite lanes. The urgent item is allowed to “jump the queue” ahead of other work that is ready to be pulled. If a team member is available, the expedite item is pulled immediately. If there is no capacity, the team can choose to block items that are already in progress to free up a team member to work on the urgent item. The cost for normal work takes the form described by Harchol-Balter (2013) for non-preemptive priority scheduling. Normal items must wait not only behind items of the same priority already in the queue. They must also wait behind all urgent items that arrive while they are waiting. Therefore, their waiting time increases with both the volume of urgent arrivals and the overall load on the system.

Preemption is a preemptive policy. When an urgent item arrives, the item currently being worked on is interrupted and moved back to the waiting queue. The urgent item is then processed immediately. The interrupted item will be worked on later. Assunção et al. (2012) describe this mechanism in detail in their simulation of dispatching policies for IT service providers. The model compares the waiting and running queues. If a higher priority item is waiting and a lower priority item is currently in service, the lower priority item is returned to the waiting queue. The higher priority item is then assigned to the newly available agent. A consequence of preemption is the overhead that arises when work is continued again. The agent may need time to reconstruct the state of the interrupted task. Assunção et al. (2012) note that this overhead ranges from 5% to 40% of the total processing time of the interrupted task, and that preemption loses its advantage once this cost exceeds approximately 45%. Compared to Priority Jumping, Preemption shortens the Lead Time of urgent items more aggressively because they do not have to wait for an ongoing item to finish. The cost for normal work is equivalently higher. In preemptive scheduling, items with lower priority experience more delay because they can be repeatedly interrupted by higher priority work (Harchol-Balter, 2013).

Reserved Capacity is structurally different from the previous two policies. Rather than regulating the sequence in which items already in the system are processed, it addresses the problem before it arises by keeping a defined share of capacity unavailable for normal work. This capacity is held in reserve so that urgent items can be picked up without displacing work already in progress. Vacanti (2025) describes this approach through the concept of slack. Similar to express logistics systems, reserved free capacity is used to absorb temporary load peaks without delaying regular deliveries. In queueing theory terms, this makes Reserved Capacity a non-work-conserving policy. It withholds capacity from the server even when work items are waiting, in order to preserve availability for higher priority arrivals (Harchol-Balter, 2013). This is the defining structural difference from the other three policies, all of which are work-conserving in that they use all available capacity

2 Theoretical Background

whenever work is present. Under Reserved Capacity, the cost for regular work is not a longer delay due to being pushed aside, but rather a permanently lower effective capacity for normal items, regardless of whether any urgent items are in the system at that moment.

Strict Expedite WIP Limit keeps the expedite lane along with its priority rules, but introduces a maximum on the number of expedite items that can be active in the system at any one time. Vacanti (2025) describes this as a widely recommended practice, with the WIP limit for the expedite lane often set to one. The intention is to prevent the expedite class from expanding without limitation and to keep the resulting flow debt under control. Strict WIP limit does not solve the underlying predictability problem. This can be shown by using the *getKanban* simulation game as an example, Vacanti (2025) reports that even with an expedite WIP limit of one, expedite items consistently finish in three days or less while standard items show a Cycle Time spread between three and fifteen days at the *P85* level. The strict WIP limit therefore reduces how often disruptions can occur, but it does not change the mechanism that causes them. In queueing theory terms, this policy remains non-preemptive and work-conserving. It differs from Priority Jumping only because it limits how many urgent items can have priority at the same time.

	Work-conserving <i>All available capacity is used</i>	Non-work-conserving <i>Capacity is intentionally withheld</i>
Non-preemptive <i>Ongoing work is not interrupted</i>	Priority Jumping <i>Urgent items jump the queue. Ongoing work finishes first</i> Strict Expedite WIP Limit <i>Same logic, but active expedite items are limited</i>	Reserved Capacity <i>A share of capacity is permanently held free for urgent arrivals</i>
Preemptive <i>Ongoing work can be interrupted</i>	Preemption <i>Urgent items displace work currently in progress</i>	<i>Not applicable: preemption already frees capacity immediately, making reservation redundant</i>

Figure 2.4: The four expedite policies along two structural dimensions. Own illustration, based on Vacanti (2025) and Harchol-Balter (2013).

These four policies differ in two key structural dimensions. The first is whether ongoing normal work can be interrupted. Preemption allows this, while the other three do not. The second is whether the policy is work-conserving. Reserved Capacity is the only one that intentionally withholds capacity, while the remaining three use all available capacity

whenever work is present. Together, these dimensions define the range of trade-offs that the following sections inspect. Figure 2.4 summarizes this classification.

2.5.6 Limits of the concept

The *Open Guide to Kanban* clearly presents classes of service as an option rather than as part of the core of Kanban ('Open Guide to Kanban', 2025). At the same time, the guide warns of two typical problematic developments. First, incorrect use can slowly overload the system. This can happen, for example, when an expedite lane slowly turns into a "super-expedite lane" and classification is used too inflationary. Second, even when classes of service are used in a formally correct way, they can still lead to an unbalanced flow. For this reason, the guide recommends using classes of service primarily as a decision aid for replenishment rather than as an instrument to prioritize work already in progress. Vacanti (2025) makes a similar point. In processes where predictability already works well, classes of service are not necessarily needed. The more stable alternative at the system-level is the intentional maintenance of reserve capacity, that is, *slack*. By analogy to logistics systems such as FedEx, slack is understood as a buffer that helps handle unexpected load peaks without damaging regular delivery promises. The *Open Guide to Kanban* applies this idea to Kanban by describing buffer capacity as an intentional part of workflow design, for example by limiting WIP to a defined share of the available capacity ('Open Guide to Kanban', 2025). In this sense, slack becomes a structural alternative to aggressive prioritization.

2.5.7 Implications for this thesis: classes of service

These four policies were chosen for comparison in this thesis because they represent four practical options for handling urgent items in a Kanban system. A team can allow urgent items to jump the queue, interrupt ongoing work, reserve capacity for urgent work, or limit the number of urgent items that can be processed at the same time. Furthermore, each policy is documented in the main Kanban literature as a typical or recommended way to handle expedite work (Kanban University, 2022; 'Open Guide to Kanban', 2025; Vacanti, 2025). Together, they cover the full set of options described in queueing theory along two dimensions: preemptive or non-preemptive, and work-conserving or non-work-conserving (Harchol-Balter, 2013). This selection gives a full and interpretable policy matrix. The last theoretical combination is not meaningful, since preemption already releases capacity. Choosing these four also makes the comparison directly relatable to earlier simulation work on WIP limits (Anderson et al., 2011), preemption with overhead (Assunção et al., 2012), and priority-based dispatching (Abdulahi, 2023), which so far have been studied separately.

The framework for service classes explained in this section is essential for this thesis. The four policies can be understood as different forms of CoS implementation. They differ in how they regulate sequencing and capacity allocation for urgent work and structure

2 *Theoretical Background*

the trade-off between fast expedite handling and the predictability of normal work. The simulation evidence provided by Vacanti (2025) provides a basis for the evaluation in this thesis. The changes in Cycle Time distributions and predictability metrics in the analysis are compared against this theoretical background. The simulation results reported in Vacanti (2025) confirm that the effects occur in the expected direction. The empirical sections investigate whether, and to what degree, these effects arise under the specific conditions of the policies and system configurations. In the rest of this thesis, the policies that define the expedite class of service are called urgent-work policies, because they specify how urgent-work items are treated compared to regular work.

3 Method

This thesis combines two components to compare policies for handling urgent work in a Kanban-based workflow. First, data from a public Jira repository are used to describe the issue flow. These include arrival patterns, completion logic, and duration measures by urgency class. Second, these inputs are used in a simulation model in which four expedite policies are compared under controlled conditions.

The four policies under comparison are:

- **Priority Jumping**, in which urgent items move to the front of the queue without interrupting Work in Progress.
- **Preemption**, in which urgent items can interrupt ongoing work.
- **Reserved Capacity**, in which a fixed share of capacity is held exclusively for urgent arrivals.
- **Strict Expedite Work-in-Progress (WIP) Limit**, in which there is a limit on how many urgent items can be active at the same time.

The policies differ in how capacity is shared between urgent and normal work. This makes the trade-off between responsiveness and predictability visible in the simulation results. The evaluation covers two dimensions. For normal work, predictability is measured through Lead Time percentiles and variability. For urgent work, performance is measured as the share of items completed within the urgent-work threshold.

The chapter is organized as follows. Section [3.1](#) describes how the empirical inputs are derived from a public Jira repository. Section [3.2](#) describes the simulation model and its assumptions. Section [3.3](#) lists the parameter values used in the runs.

3.1 Data Preparation

This section describes how the simulation inputs are derived from the Public Jira Dataset (Montgomery et al., [2025](#)). The inputs are introduced one at a time: the dataset and time window, the classification of work items, completion and Lead Time, arrivals, and service time. A short summary at the end lists the three inputs that are inserted into the simulation.

3.1.1 Dataset

The Public Jira Dataset (Montgomery et al., 2025) is a publicly available collection of data from 16 public Jira repositories, covering 1,822 software projects and around 2.7 million issues. It contains real-world data of how work items are created, prioritized, moved through workflow states, and resolved. This thesis uses the SPARK project from the Public Jira Dataset as the basis for the simulation. SPARK is the Jira repository of Apache Spark, an open-source software project used to process large amounts of data. SPARK was selected because the project provides a sufficiently large and complete Jira case for the intended simulation setup. In the extracted data, SPARK contains 37,443 issues, of which 35,133 have a usable completion status and 2,310 are unfinished. The completed set contains 32,253 normal and 2,880 urgent issues. This makes the project large enough to derive arrival patterns, Lead Time distributions, and tail measures such as P85 and P95 for both normal and urgent work. Earlier data checks also showed broad status-history availability, with 35,994 issues containing status history. SPARK was therefore suitable not only for calculating average behavior, but also for modelling variability and predictability.

The analysis and setup window is restricted to 2015–2018 because the derived `work_hours` variable, which serves as the service-time baseline in the simulation, has substantially better coverage in this period than in the remaining SPARK data. Across the full completed SPARK dataset, only about 58.5% of issues have usable `work_hours` values, while the 2015–2018 window reaches about 68.5%. In comparison, the remaining SPARK data outside this window reach only about 47.0%. The difference is especially important for urgent work, where coverage is about 65.4% in 2015–2018 but only about 38.3% outside this window. The window choice is therefore driven by data suitability rather than convenience. At the same time, the chosen window stays large enough to support an empirical setup. For 2015–2018, the data contain 210 weekly observations, with 19,492 normal and 1,910 urgent arrivals. The raw Jira data contain issue-level records and status transition records. From the issue table, the thesis uses the issue key, creation timestamp, resolution timestamp, issue type, and priority. From the status history, transitions between workflow states are used to reconstruct completion information and duration.

3.1.2 Data cleaning and classification

The raw SPARK files were not used directly as simulation input. Before constructing the inputs, the data preparation pipeline checked the required issue and status-event files, parsed the relevant timestamp fields, and treated empty timestamp strings as missing values. Malformed year values with a leading 00 were corrected before parsing, because these dates could otherwise not be interpreted consistently. The analysis then distinguishes between urgent and normal work. Classification uses Jira priority as the operational proxy. Issues with priority `Blocker` or `Critical` are classified as urgent, and all remaining issues are classified as normal. The cleaning process then separated

3 Method

the data according to the input that had to be created. Arrival inputs were based on `created_at` and restricted to issues created in the 2015–2018 window. Weekly arrival counts were merged with a complete class-week grid. Missing counts were then set to zero, so that each week in the 2015–2018 window is represented for both normal and urgent work.

For completion-based measures, only issues with a usable creation timestamp and a usable final completion timestamp were kept. The final completion timestamp was constructed as described in Section 3.1.3. Status-transition records were used only when they had a valid transition timestamp and belonged to completed issues in the analysis window. Completed issues without usable status history were kept for completion-based measures, but their missing workflow history was made explicit through a synthetic `NoStatusHistory` stay. The service-time input was cleaned separately. Since the simulation uses `work_hours` as service-time proxy, only observations with usable positive `work_hours` were included in the empirical service-time sample. The baseline input uses this cleaned sample without capping. A second input, Trim995, was created by capping `service_hours` at the 99.5th percentile within each work class. This second input is used only for the service-time sensitivity analysis. Finally, the generated outputs were checked with a validation script. The script compares the reproduced arrival, completion, and service-time outputs with the reference values used in this thesis. This links the method chapter to the reproducibility material in Appendix A without moving the full technical pipeline into the main text.

3.1.3 Completion and Lead Time

A completion timestamp is needed to identify completed issues and to extract their durations. Completion is therefore constructed in two steps. If a valid `resolutiondate` is available, it is used as the primary completion timestamp. If `resolutiondate` is missing, a fallback timestamp is derived from the last transition into a done state. This procedure makes sure that completed issues are kept even when `resolutiondate` is missing, as long as the status history contains a usable final transition into a done status. Lead Time is measured from issue creation to the final completion timestamp. Table 3.1 reports the resulting Lead Time distribution for completed normal and urgent issues in the 2015–2018 window.

Class	<i>n</i>	Mean	P50	P85	P95	P99	Min	Max
Normal	16,850	1,753.97 h	158.88 h	3,672.05 h	10,290.47 h	18,086.51 h	0.00 h	47,541.31 h
Urgent	1,976	1,479.62 h	148.79 h	2,281.06 h	8,262.92 h	21,820.13 h	0.02 h	40,197.46 h

Table 3.1: Lead Time distribution for completed normal and urgent issues.

For the urgent class, the Lead Time distribution contains 1,976 completed issues, with a mean of 1,479.62 h, a median of 148.79 h, a P85 of 2,281.06 h, and a P95 of 8,262.92 h. The P85 value of 2,281.06 h, rounded to 2,281 h, is used as the urgent-work threshold in

3 Method

the simulation. This aligns the urgent performance benchmark with the window used for arrivals and service-time setup. The urgent Lead Time validation sample is not identical to the urgent arrival series. The arrival series is defined by urgent issues created in 2015–2018 ($n = 1,910$), whereas the Lead Time validation sample is defined by completed urgent issues with `done_at_final` in 2015–2018 ($n = 1,976$). The overlap between both sets is 1,849 urgent issues. The 127 urgent issues outside of the overlap were created before 2015, but completed within the 2015–2018 window. They are included in the completed Lead Time sample, but not in the arrival series, which counts only issues created in 2015–2018.

For the normal class, the minimum reported Lead Time is 0.00 h. A closer look at this case shows that the value results from a very short but non-zero recorded Lead Time rather than from a truly zero-duration case. Specifically, issue **SPARK-8299** was created on 2015-06-11 00:42:24 and reached its final completion timestamp on 2015-06-11 00:42:37, resulting in a Lead Time of approximately 13 seconds (0.0036 h). This case is therefore reported as a lower end extreme in the distribution.

3.1.4 Arrivals

Arrivals are derived from `created_at` and summarized as weekly counts per class. For the 2015–2018 window, this results in 210 weeks, with 19,492 normal and 1,910 urgent arrivals. Table 3.2 reports the weekly averages and standard deviations.

Class	Total arrivals	Mean arrivals per week	SD arrivals per week
Normal	19,492	92.82	34.18
Urgent	1,910	9.10	10.07
Total	21,402	101.91	41.59

Table 3.2: Arrival summary.

Arrivals are aggregated on a weekly basis instead of a more detailed time scale. The urgent class only has 1,910 arrivals over four years, which is about 1.3 per day. At a daily level, this would mostly produce zero-arrival days. At a weekly level, there are 210 stable observations with a mean of 9.10 urgent arrivals per week. This is consistent with Kanban systems in practice, including the use of replenishment meetings and the time unit that Vacanti (2025) uses to analyze arrival and departure rates. These weekly class-specific counts form the arrival input for the simulation.

3.1.5 Service time

In line with the concept of queueing theory of service time introduced in Section 2.2.1, service time is operationalized through `work_hours` and treated as the simulation duration input. Specifically, `work_hours` is derived from observed time in the workflow states `In Progress` and `Reopened`. This variable is not treated as a direct measurement of actual work effort. It serves as the best available operational proxy for service time in this window. The service-time input is prepared in two layers. First, a cleaning step is applied to `work_hours` in all cases (removing unusable observations and producing the baseline service-time sample). Second, an optional capping step is applied only for the Trim995 variant. The baseline service-time input uses the cleaned `work_hours` without capping. The `work_hours` sample is a subset of the issues in the 2015–2018 window, because only cases with usable work information are considered for the service time distribution. After restricting the data to usable `work_hours` and applying the cleaning step, the baseline service-time input contains 11,609 normal and 1,293 urgent observations.

Measure	Normal	Urgent
Baseline service time		
Sample size after cleaning	11,609	1,293
Mean (h)	845.20	557.54
P50 (h)	93.64	50.35
P85 (h)	1,015.81	557.94
P95 (h)	4,883.56	2,508.17
Maximum (h)	27,238	29,519
Trim995 service time		
Cap value (h)	15,280.94	15,246.60
Capped observations	59	7
Share of capped observations (%)	0.51	0.54
Mean after capping (h)	826.72	521.59
Maximum after capping (h)	15,281	15,247

Table 3.3: Service time summary: baseline and Trim995.

Because the distribution contains extreme right tail values, a separate Trim995 input variant is prepared for the service time sensitivity analysis in Section 4.3. In Trim995, values above the class-specific 99.5th percentile are capped after cleaning. Table 3.3 reports the baseline service-time input together with the additional capping information for the Trim995 input. In the simulation, the service time of each item is sampled from the class-specific `work_hours` distribution after cleaning. Service times and resulting measures remain in hours.

3.1.6 Summary of simulation inputs

The simulation uses three inputs derived from SPARK 2015–2018: weekly class-specific arrivals (see Table 3.2), class-specific `work_hours` service-time samples (see Table 3.3), and an urgent-work threshold of 2,281 h. Arrivals, the urgent-work threshold, and the service-time proxy are all derived from the same window. This avoids mixing statistics from different project periods and allows a consistent setup. All policy comparisons in the following sections use these inputs.

3.2 Simulation Model

The model was implemented in AnyLogic 8.9.6 (The AnyLogic Company, 2025) as a discrete-event simulation of a Kanban system, with two work classes (normal and urgent). Its purpose is to compare how the four expedite policies affect the predictability of normal work and the performance of urgent work. Rather than reproducing every detail of a real Jira workflow, the model represents only the system logic required for the policy comparison: arrivals, queueing, dispatching, service, interruption, and completion.

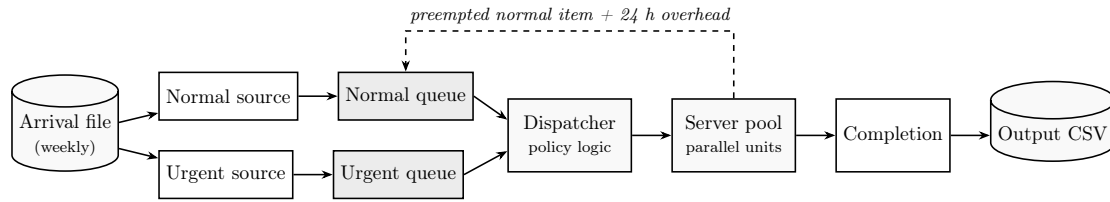


Figure 3.1: Simulation model flow (AnyLogic).

3.2.1 Agents and components

The top-level agent is **Main**, which contains the model components: arrival elements, queues, completion logic, input handling, output writing, and the server population. Two additional agent types are used. The first is **WorkItem**, which represents a single unit of work. Each work item saves the details required during the simulation: class (normal or urgent), arrival time, sampled service time, remaining service time, completion time, and, for urgent items, the urgent-work threshold against which Lead Time is compared. The second type of agent is **Server**, which represents one unit of processing capacity. Each server tracks whether it is currently busy and which work item it is processing. Under the Reserved Capacity policy, it also tracks whether it is reserved for urgent work.

3.2.2 Arrival handling

Arrivals enter the model through two source elements, one per work class. Instead of being driven by an internal rate distribution, both sources are triggered from an external arrival file. A weekly event reads one line from this file and inserts the matching number of normal and urgent items into the model. After creation, each item is assigned a sampled service time from the class-specific `work_hours` distribution. Normal and urgent items then enter separate queues. The separation is required because dispatching rules depend on the work class and the active policy.

3.2.3 Dispatching and policy logic

The model uses its own custom routing logic instead of the standard AnyLogic service block and its integrated priority settings. This design is required because the four policies rely on different routing and selection rules that cannot be represented within a single block configuration. Whenever a work item enters a queue or a server becomes idle, the dispatching function checks whether a free server is available and which item should be processed next.

The four policies differ only in this dispatching logic:

- Under **Priority Jumping**, urgent items are selected before normal items, but ongoing work is not interrupted.
- Under **Preemption**, an urgent arrival may interrupt a normal item currently in service.
- Under **Reserved Capacity**, a defined share of servers is reserved exclusively for urgent items.
- Under **Strict Expedite WIP Limit**, urgent items may only enter service if the number of urgent items currently in service is below the defined limit.

This logic is implemented through dedicated functions rather than separate model branches, so that the system structure stays the same across all policy runs.

3.2.4 Service and completion

Service is controlled at the server level. Each server owns a dedicated completion event. When service begins, the assigned server stores a reference to the item, marks itself busy, and schedules its completion event for the item's remaining service time. If preemption occurs, the interrupted item is removed from the server, its remaining service time is updated, a preemption overhead is added, and the item reenters the queue. When a completion event is triggered, the model records the item's completion, calculates its Lead

Time, updates the corresponding result lists, and, for urgent items, checks whether the item was completed within the urgent-work threshold. The server is then released, and the dispatching logic is triggered again.

3.2.5 Termination and outputs

The simulation stops only when the defined termination conditions are met. A run ends only when three conditions are met at the same time: all arrivals have been loaded, all queues are empty, and all servers are idle. This guarantees that each run processes the complete arrivals before any results are written. The Lead Times for normal and urgent work are recorded separately, and the summary measures are exported to a CSV file at the end of each run. The main output metrics are the Lead Time percentiles for normal work (P50, P85, P95) and the share of urgent items completed within the urgent-work threshold. The basic process structure stays the same across all runs. Only the dispatching and service logic change depending on the policy.

3.2.6 Model assumptions

The simulation uses the SPARK project from the Public Jira Dataset (Montgomery et al., 2025) and focuses on 2015–2018. This window was chosen because the derived `work_hours` variable, which serves as the service-time baseline, has sufficient coverage only in this period. Within this time frame, Jira entries are assumed to offer a consistent and reliable record of the variables used in this thesis. Specifically, `created_at` provides the arrival signal, priority classifies issues as urgent or normal, and status and resolution information are used to reconstruct completion and duration data. The recorded events are usable for the reconstruction and simulation setup used in this thesis. Arrivals are inserted into the model on a weekly basis, service times and resulting measures remain in hours. A model hour is one unit on the empirical time scale used for the setup. It is not the same as a unit of actual labor time.

Service time is measured with the `work_hours` variable. This variable is used as the service time baseline for data preparation, parameterization, and simulation. `work_hours` does not directly measure processing effort. For this period, `work_hours` is the best available approximation of service time. `total_hours` also includes time in states that do not represent active work. Using it would change the meaning of service time. For this reason, the model uses only `work_hours`. Capacity is represented in an abstract way. The servers in the simulation are not individual people from the Jira project. Instead, they are parallel capacity units that process work items. This is required because the model compares expedite policies under controlled conditions. It does not represent the actual team setup of the SPARK project.

3 Method

The same basic system is used for all four expedite policies. The policies differ only in their queueing rules. Priority Jumping is a non-preemptive priority rule. Preemption is a preemptive priority rule. Reserved Capacity is a capacity allocation rule. Strict Expedite WIP Limit is a rule that limits the number of urgent items allowed in service at the same time. A preemption penalty is included only as an assumption for the preemption scenario. Another assumption is about capacity scaling. When total system capacity changes across scenarios, the relevant policy constraints scale with it. This applies to reserved capacity shares and expedite WIP limits. Without such scaling, the policy differences would shrink as capacity grows, and the comparison would no longer reflect the intended policy logic. The outputs are based on this setup. For normal work, predictability is measured with Lead Time percentiles (P50, P85, P95). For urgent work, the model reports urgent Lead Times and the share of urgent items completed within the 2,281-hour threshold.

These assumptions also define the limits of the model. The simulation is set up on a single Jira project and has limited external validity. The distinction between urgent and normal work is based on Jira priority values, not on observation of how critical each issue was to the organization. Arrivals are inserted on a weekly basis and do not take bursts during the week into account. `work_hours` remains a derived proxy, not a direct observation of actual work time. The model results should be read as a comparison of policy outcomes within this framework. They are not exact forecasts of real-world performance.

3.3 Parameterization

The model structure described in Section 3.2 requires specific parameters before it can be run. This section lists these values and explains how they were chosen. Some parameters are derived directly from the SPARK data, while others are design choices. Arrivals, the urgent-work threshold, and the class-specific `work_hours` distributions are derived from the 2015–2018 window. The capacity level, the Reserved Capacity and Strict Expedite WIP Limit settings, and the preemption overhead are design parameters. They are not estimated from SPARK, but set to values that make the policy comparison meaningful under controlled conditions. The urgent threshold is set to 2,281 h (see Section 3.1).

3.3.1 Capacity

The main runs use a capacity of 1,000 servers. This value is not intended to represent the number of developers in the SPARK project or in a typical Kanban team. It is a parameter that represents the abstract processing capacity of the simulation. The value was chosen after exploratory runs with smaller capacities, where queue sizes and Lead Times grew strongly and did not produce stable, comparable results across policies. A capacity of 1,000 stabilizes the system enough to compare policies. The robustness scenarios use 800 and 1,200 servers to test whether the baseline findings hold under tighter and looser

3 Method

capacity assumptions. When total capacity changes, the Reserved Capacity share and the Strict Expedite WIP Limit are adapted accordingly. Without this adaptation, the policy constraints would shrink as capacity grows, and the comparison would no longer represent the policy logic.

3.3.2 Policy parameters

Three of the four policies require additional parameters. Priority Jumping does not, because it only reorders the queue and does not interrupt ongoing work.

Under **Preemption**, a fixed overhead of 24 h is added to the remaining service time of an interrupted item before it reenters the queue. This value represents the cost of interruption and resumption (see Section 2.4.2). Compared to the median normal `work_hours` of 93.64 h, 24 h represents around 26%. This falls within the 5–40% range reported by Assunção et al. (2012), and below the 45% threshold above which preemption stops being beneficial.

Under **Reserved Capacity**, 30 servers are reserved exclusively for urgent work in the baseline scenario. This represents 3% of the total capacity, which is below the urgent share of around 9% of arrivals recorded in SPARK. This setting keeps most of the capacity available for normal work, while still leaving a small buffer for urgent items. This reflects the principle of slack discussed in Section 2.4.4. The reserved share is kept at 3% in all scenarios, because varying this share is outside the scope of this thesis.

Under **Strict Expedite WIP Limit**, a maximum of 30 urgent items may be in service at the same time. This value was set equal to the Reserved Capacity parameter to keep both policies comparable, as they are capacity restricting.

In the robustness scenarios, the Reserved Capacity and the Strict Expedite WIP Limit are adapted with total capacity. They are set to 24 for 800 servers and 36 for 1,200 servers.

3.3.3 Replications and termination

Each policy is run with 10,000 independent replications, each with a different random seed. Many replications are needed because P85 and P95 can vary strongly between runs. A large number of runs makes these tail estimates more reliable. The 10,000 runs per policy make it possible to report the central value of each metric and its variability across runs. This makes it possible to carry out the analysis in Section 4, where policies are compared not only in terms of their median results, but also with respect to the variability of those results across runs.

3 Method

There is no fixed stop time. Each run ends when all arrivals have been loaded, all queues are empty, and all servers are idle. This makes sure that every run processes the complete arrivals before results are written. Table 3.4 summarizes the parameters.

Parameter	Value
Simulation window	2015–2018
Arrivals	Weekly, class-specific
Service-time input	<code>work_hours</code> , cleaned
Service-time sensitivity input (Trim995)	<code>work_hours</code> , cleaned & capped
Urgent threshold	2,281 h
Preemption overhead	24 h
Capacity	1,000 servers (baseline); 800 / 1,200 (robustness)
Reserved Capacity	30 servers (baseline); 24 / 36 (robustness)
Strict Expedite WIP Limit	30 items (baseline); 24 / 36 (robustness)
Replications per policy	10,000

Table 3.4: Parameters for the simulation.

4 Results

This section reports the simulation results for the four urgent-work policies introduced in Section 2.5.5. The policies were implemented in AnyLogic 8.9.6 and ran under the setup described in Section 3. Each policy was executed 10,000 times, each run with a different random seed. This results in 160,000 runs in total. Each run ended only when all arrivals had been loaded, all queues were empty, and all servers were idle. The section is structured around the two outcome dimensions defined in Section 3. For normal work, predictability is reported through Lead Time percentiles (P50, P85, P95) and their run-to-run variability. For urgent work, performance is reported through urgent Lead Time percentiles and the urgent hit rate. A run-level summary then reports the finish time of each run, the number of urgent items completed within the threshold, and the number of preemption events.

The results follow two reporting principles. The first is that the values shown in the main tables are the median across the 10,000 runs per policy. The median is chosen over the mean because it is less sensitive to outliers and because the distribution of each metric is close to symmetric, which means the two values are close in practice. Second, variability across runs is reported in separate tables. These report the variation and the P5 and P95 values of each metric. The variation is reported as a percentage and measures the standard deviation relative to the mean. This makes the stability of different metrics comparable, even when their absolute levels differ. The range P5 to P95 describes where the middle 90% of run-level outcomes fall and is used throughout the section to compare whether policy ranges overlap. In addition to the per-metric tables, a specific measure is used to describe the shape of the normal-work Lead Time distribution. The spread is measured as the difference between P95 and P50. It shows how strongly Lead Times increase in the upper part of the distribution. A policy with a smaller spread results in a more compact Lead Time distribution and makes regular work more predictable, regardless of the specific percentile values.

Section 4.1 presents the baseline scenario at 1,000 servers. Section 4.2 presents the robustness scenarios on 800 and 1,200 servers and closes with a summary of all three capacity levels. Section 4.3 presents a service-time sensitivity analysis that re-runs the baseline scenario with extreme right tail values capped at the 99.5th percentile (Trim995).

4.1 Baseline scenario

The baseline scenario uses 1,000 servers and the same setup described in Section 3. It uses the full service-time input without the 99.5% trimming variant. The urgent-work threshold is set to 2,281 h, the preemption overhead to 24 h, the Reserved Capacity parameter to 30 servers, and the Strict Expedite WIP Limit parameter to 30 urgent items in service. All four policies process the same arrivals and the same service-time inputs. The total number of completed items is 21,402, consisting of 19,492 normal items and 1,910 urgent items. This means the performance metrics reported below can be compared across policies.

4.1.1 Normal work

Table 4.1 shows the normal-work predictability metrics for the baseline scenario. Strict Expedite WIP Limit produced the lowest normal-work percentile levels, with a median P50 of 5,461 h, a median P85 of 11,356 h, and a median P95 of 13,057 h. Reserved Capacity produced the highest percentile levels, with a median P50 of 6,388 h, a median P85 of 12,369 h, and a median P95 of 14,067 h. Priority Jumping and Preemption stayed close to each other at all three percentile levels, with differences below 60 h.

Policy	P50 (h)	P85 (h)	P95 (h)	P95–P50 (h)
Priority Jumping	6,143	11,937	13,594	7,446
Preemption	6,130	11,976	13,646	7,513
Reserved Capacity	6,388	12,369	14,067	7,678
Strict Expedite WIP Limit	5,461	11,356	13,057	7,596

Table 4.1: Baseline: normal work predictability.

Priority Jumping shows the lowest median spread at 7,446 h, followed by Preemption at 7,513 h and Strict Expedite WIP Limit at 7,596 h. Reserved Capacity shows the highest spread at 7,678 h. The ordering by percentile level does not match the ordering by spread. Strict Expedite WIP Limit produces the lowest percentile levels, but only the third lowest spread. Priority Jumping shows the lowest spread, but not the lowest percentile levels.

Table 4.2 reports how stable these results are across the 10,000 runs per policy. For all four policies, the mean across runs is almost equal to the median, which indicates that the run-level distribution of each metric is close to symmetric. The standard deviations are small relative to the level of each metric. For Normal P50, the standard deviation ranges from 193 h to 211 h across policies, which corresponds to a variation of about 3.2% to 3.5%. For Normal P85 and Normal P95, the variations are even lower, at about 2.5% and 2.3%. Overall, the four policies show comparable run-to-run stability.

4 Results

Policy	Metric	Median (h)	SD (h)	P5 (h)	P95 (h)
Priority Jumping	Normal P50	6,143	202	5,829	6,493
Preemption	Normal P50	6,130	198	5,805	6,460
Reserved Capacity	Normal P50	6,388	211	6,051	6,741
Strict Expedite WIP Limit	Normal P50	5,461	193	5,145	5,788
Priority Jumping	Normal P85	11,937	301	11,455	12,448
Preemption	Normal P85	11,976	295	11,494	12,462
Reserved Capacity	Normal P85	12,369	307	11,866	12,875
Strict Expedite WIP Limit	Normal P85	11,356	293	10,889	11,855
Priority Jumping	Normal P95	13,594	316	13,090	14,135
Preemption	Normal P95	13,646	310	13,141	14,154
Reserved Capacity	Normal P95	14,067	321	13,541	14,596
Strict Expedite WIP Limit	Normal P95	13,057	305	12,574	13,573
Priority Jumping	Spread	7,446	219	7,093	7,813
Preemption	Spread	7,513	219	7,161	7,887
Reserved Capacity	Spread	7,678	224	7,307	8,049
Strict Expedite WIP Limit	Spread	7,596	221	7,242	7,969

Table 4.2: Baseline: run-level variability of normal work.

The P5 and P95 columns describe the range of each metric across runs. For Normal P50, the ranges are 5,829 h to 6,493 h for Priority Jumping, 5,805 h to 6,460 h for Preemption, 6,051 h to 6,741 h for Reserved Capacity, and 5,145 h to 5,788 h for Strict Expedite WIP Limit. The ranges of Priority Jumping and Preemption overlap almost completely. The range of Reserved Capacity is shifted upward and partly overlaps with both. The range of Strict Expedite WIP Limit lies clearly below all three other policies, without overlap. At Normal P85 and Normal P95, the ordering of policies is the same, but the ranges of Strict Expedite WIP Limit partly overlap with the lower ends of Priority Jumping and Preemption.

4.1.2 Urgent work

Table 4.3 reports the urgent-work metrics. Three of the four policies achieved similar results, Strict Expedite WIP Limit clearly separated from the other three. Preemption achieved the shortest urgent Lead Times, with a median P50 of 50.3 h, a median P85 of 558 h, and a median P95 of 2,510 h. Priority Jumping and Reserved Capacity are almost identical. Their median P50 values are 93.5 h and 93.4 h, their median P85 values are 592 h and 591 h, and their median P95 values are 2,548 h and 2,549 h. The median urgent hit rate is 94.5% for Preemption, 94.3% for Priority Jumping, and 94.3% for Reserved Capacity.

4 Results

Policy	Urgent hit rate (%)	Urgent P50 (h)	Urgent P85 (h)	Urgent P95 (h)	Total preemptions
Priority Jumping	94.3	93.5	592	2,548	0
Preemption	94.5	50.3	558	2,510	1,722
Reserved Capacity	94.3	93.4	591	2,549	0
Strict Expedite WIP Limit	13.4	13,963	26,369	29,879	0

Table 4.3: Baseline: urgent work performance.

Strict Expedite WIP Limit performed the worst on all urgent-work measures. Its median urgent hit rate is 13.4%. The urgent Lead Time percentiles are 13,963 h at P50, 26,369 h at P85, and 29,879 h at P95. Preemption events only occurred under the Preemption policy, with a median count of 1,722 per run. The other three policies have zero preemption events by design. Priority Jumping does not interrupt ongoing work. Reserved Capacity sets aside capacity for urgent items without interrupting normal items. Strict Expedite WIP Limit only restricts how many urgent items can be in service at the same time.

For all four policies, the mean between runs is close to the median, which indicates that the run-level distribution of each metric is close to symmetric (Table 4.4). The urgent Lead Time percentiles are less stable across runs than the normal-work percentiles. For Preemption, Priority Jumping, and Reserved Capacity, the variation is about 6% to 8% for Urgent P50, and about 8% to 9% for Urgent P85 and Urgent P95. Preemption has the smallest standard deviation for Urgent P50, at 3.05 h, while the values for Priority Jumping and Reserved Capacity are about 7.3 h. The urgent hit rate is the most stable metric across runs for these three policies. Its standard deviation is about 0.005, which corresponds to a variation below 1%. Strict Expedite WIP Limit is the least stable policy on the urgent hit rate, with a variation of about 13%, compared to below 1% for the other three policies.

For Urgent P50, the ranges are 46.1 h to 55.7 h for Preemption, 81.7 h to 106.0 h for Priority Jumping, 81.6 h to 106.0 h for Reserved Capacity, and 11,300 h to 17,200 h for Strict Expedite WIP Limit. The ranges of Priority Jumping and Reserved Capacity overlap almost completely, the range of Preemption lies clearly below both without overlap, and the range of Strict Expedite WIP Limit is in a different order of magnitude. For Urgent P85 and Urgent P95, the policy ranking is the same. However, the ranges of Preemption, Priority Jumping, and Reserved Capacity overlap. At Urgent P85, the P5–P95 range runs from 482 h to 637 h for Preemption, from 517 h to 679 h for Priority Jumping, and from 516 h to 679 h for Reserved Capacity. At Urgent P95, all three ranges overlap heavily. For the urgent hit rate, the ranges of Preemption (93.6% to 95.3%), Priority Jumping (93.5% to 95.2%), and Reserved Capacity (93.5% to 95.2%) are almost identical. Strict Expedite WIP Limit has a range from 10.5% to 16.4%, without overlap with any other policy.

4 Results

Policy	Metric	Median	SD	P5	P95
Priority Jumping	Urgent P50 (h)	93.5	7.37	81.7	106
Preemption	Urgent P50 (h)	50.3	3.05	46.1	55.7
Reserved Capacity	Urgent P50 (h)	93.4	7.31	81.6	106
Strict Expedite WIP Limit	Urgent P50 (h)	13,963	1,780	11,300	17,200
Priority Jumping	Urgent P85 (h)	592	49.5	517	679
Preemption	Urgent P85 (h)	558	49.5	482	637
Reserved Capacity	Urgent P85 (h)	591	49.7	516	679
Strict Expedite WIP Limit	Urgent P85 (h)	26,369	2,610	22,300	30,800
Priority Jumping	Urgent P95 (h)	2,548	217	2,220	2,920
Preemption	Urgent P95 (h)	2,510	219	2,160	2,890
Reserved Capacity	Urgent P95 (h)	2,549	220	2,220	2,930
Strict Expedite WIP Limit	Urgent P95 (h)	29,879	2,780	25,500	34,700
Priority Jumping	Urgent hit rate (%)	94.3	0.5	93.5	95.2
Preemption	Urgent hit rate (%)	94.5	0.5	93.6	95.3
Reserved Capacity	Urgent hit rate (%)	94.3	0.5	93.5	95.2
Strict Expedite WIP Limit	Urgent hit rate (%)	13.4	1.8	10.5	16.4

Table 4.4: Baseline: run-level variability of urgent work.

4.1.3 Run-level summary

Policy	Total completed	Completed urgent	Urgent met threshold	Simulation finish time (h)
Priority Jumping	21,402	1,910	1,802	37,610
Preemption	21,402	1,910	1,804	37,667
Reserved Capacity	21,402	1,910	1,802	38,078
Strict Expedite WIP Limit	21,402	1,910	255	53,233

Table 4.5: Baseline: completion and run-level summary.

Table 4.5 reports the run-level comparison. All four policies completed the full workload, with a median of 21,402 completed items and 1,910 completed urgent items. The median number of urgent items completed within the 2,281 hour threshold is 1,804 for Preemption, 1,802 for both Priority Jumping and Reserved Capacity, and 255 for Strict Expedite WIP Limit. The simulation finish time shows the same split between policies. Priority Jumping and Preemption finished close together, with median finish times of 37,610 h and 37,667 h. Reserved Capacity finished 468 h later at 38,078 h. Strict Expedite WIP Limit finished clearly last at 53,233 h, 15,623 h after Priority Jumping. In addition to the median comparison, Table 4.6 shows the variability of the run-level metrics that changed between runs. The completed workload itself is constant across runs for all four policies and is therefore not included in this table.

4 Results

Policy	Metric	Median	SD	P5	P95
Priority Jumping	Finish time (h)	37,610	1,720	34,855	40,524
Preemption	Finish time (h)	37,667	1,720	34,928	40,555
Reserved Capacity	Finish time (h)	38,078	1,750	35,279	41,008
Strict Expedite WIP Limit	Finish time (h)	53,233	5,950	42,358	61,690
Priority Jumping	Urgent items met threshold	1,802	10.1	1,785	1,818
Preemption	Urgent items met threshold	1,804	10.0	1,787	1,820
Reserved Capacity	Urgent items met threshold	1,802	10.2	1,785	1,819
Strict Expedite WIP Limit	Urgent items met threshold	255	34.3	201	314
Priority Jumping	Preemptions	0	0	0	0
Preemption	Preemptions	1,722	0.49	1,722	1,722
Reserved Capacity	Preemptions	0	0	0	0
Strict Expedite WIP Limit	Preemptions	0	0	0	0

Table 4.6: Baseline: run-level variability of completion and preemption.

The additional statistics show that the same pattern stays visible between runs. Priority Jumping and Preemption stay close together on finish time, while Reserved Capacity is higher, and Strict Expedite WIP Limit is clearly higher than the other three policies. The P5 to P95 range for finish time is 34,855 h to 40,524 h for Priority Jumping, 34,928 h to 40,555 h for Preemption, 35,279 h to 41,008 h for Reserved Capacity, and 42,358 h to 61,690 h for Strict Expedite WIP Limit. The same pattern appears for the number of urgent items completed within the threshold. Priority Jumping, Preemption, and Reserved Capacity stay close together, with P5 to P95 ranges from 1,785 to 1,818, from 1,787 to 1,820, and from 1,785 to 1,819. Strict Expedite WIP Limit stays far below the other three policies, with a range from 201 to 314. Preemptions only occur under the Preemption policy. Their variability is very small, with a median of 1,722 and a standard deviation of 0.49.

4.2 Robustness scenarios

4.2.1 800 servers

Normal work

Table 4.7 reports the normal-work predictability metrics for the 800-server scenario. Strict Expedite WIP Limit produced the lowest percentile levels, while Reserved Capacity produced the highest. Priority Jumping and Preemption stayed close to each other, with differences below 70 h. The ordering by median spread is Priority Jumping (9,347 h) < Preemption (9,431 h) < Strict Expedite WIP Limit (9,542 h) < Reserved Capacity (9,627 h), and again does not match the ordering by percentile level. Standard deviations

4 Results

are small relative to the level of each metric, with variations of about 3.2% to 3.4% at P50 and about 2.2% to 2.3% at P85 and P95.

Policy	Metric	Median (h)	SD (h)	P5 (h)	P95 (h)
Priority Jumping	Normal P50	8,439	268	8,014	8,900
Preemption	Normal P50	8,429	266	8,004	8,872
Reserved Capacity	Normal P50	8,756	281	8,308	9,226
Strict Expedite WIP Limit	Normal P50	7,554	256	7,142	7,990
Priority Jumping	Normal P85	15,753	369	15,141	16,358
Preemption	Normal P85	15,801	370	15,203	16,417
Reserved Capacity	Normal P85	16,275	381	15,656	16,909
Strict Expedite WIP Limit	Normal P85	15,008	364	14,421	15,618
Priority Jumping	Normal P95	17,793	392	17,152	18,444
Preemption	Normal P95	17,860	396	17,218	18,522
Reserved Capacity	Normal P95	18,386	407	17,733	19,061
Strict Expedite WIP Limit	Normal P95	17,105	387	16,475	17,755
Priority Jumping	Spread	9,347	269	8,910	9,793
Preemption	Spread	9,431	275	8,988	9,896
Reserved Capacity	Spread	9,627	279	9,184	10,092
Strict Expedite WIP Limit	Spread	9,542	277	9,105	10,012

Table 4.7: 800-server: run-level variability of normal work.

Compared to the baseline, the median P50 rose by about 37% to 38%, the median P85 by about 32%, the median P95 by about 31%, and the median spread by about 25% to 26%. The increase is similar across policies, and the ordering of policies by percentile level and spread is unchanged.

Urgent work

Table 4.8 reports the urgent-work metrics. Three of the four policies again achieved similar results, while Strict Expedite WIP Limit clearly separated from the rest. Preemption achieved the shortest urgent Lead Times, Priority Jumping and Reserved Capacity are nearly identical, and Strict Expedite WIP Limit reached the highest medians. The median urgent hit rate is 94.5% for Preemption, 94.2% for both Priority Jumping and Reserved Capacity, and 10.7% for Strict Expedite WIP Limit. Preemption events again occurred only under the Preemption policy, with a median count of 1,787 per run, slightly higher than the baseline value of 1,722. Standard deviations are small for the first three policies (variation between about 6% and 9% on the Lead Time percentiles, below 1% on the hit rate), while Strict Expedite WIP Limit shows a variation between about 9% and 15%.

4 Results

Policy	Metric	Median	SD	P5	P95
Priority Jumping	Urgent P50 (h)	149	12.3	130	171
Preemption	Urgent P50 (h)	50.3	3.1	46.1	55.9
Reserved Capacity	Urgent P50 (h)	149	12.4	130	171
Strict Expedite WIP Limit	Urgent P50 (h)	18,400	2,340	14,900	22,600
Priority Jumping	Urgent P85 (h)	634	50.0	556	719
Preemption	Urgent P85 (h)	558	49.1	482	637
Reserved Capacity	Urgent P85 (h)	635	50.1	558	720
Strict Expedite WIP Limit	Urgent P85 (h)	33,900	3,350	28,800	39,800
Priority Jumping	Urgent P95 (h)	2,580	215	2,260	2,960
Preemption	Urgent P95 (h)	2,510	215	2,160	2,890
Reserved Capacity	Urgent P95 (h)	2,580	221	2,260	2,980
Strict Expedite WIP Limit	Urgent P95 (h)	38,300	3,540	32,800	44,500
Priority Jumping	Hit rate (%)	94.2	0.5	93.3	95.1
Preemption	Hit rate (%)	94.5	0.5	93.6	95.3
Reserved Capacity	Hit rate (%)	94.2	0.6	93.2	95.1
Strict Expedite WIP Limit	Hit rate (%)	10.7	1.6	8.2	13.5

Table 4.8: 800-server: run-level variability of urgent work.

Compared to the baseline, the results did not change in the same way for all policies, and Preemption remained almost unchanged. For Priority Jumping and Reserved Capacity, the median Urgent P50 rose from about 93.5 h to 149 h (about 59%), while their P85 and P95 rose only by about 7% and 1% respectively. Strict Expedite WIP Limit worsened, with Lead Time percentiles up by about 28% to 32% and hit rate down from 13.4% to 10.7%. The ordering of policies did not change.

Run-level summary

Table 4.9 reports the results at the run-level. All four policies again completed the full workload of 21,402 items (1,910 urgent). Priority Jumping and Preemption finished close together (median finish times of 41,765 h and 41,836 h), Reserved Capacity finished 579 h later (42,344 h), and Strict Expedite WIP Limit finished clearly last at 60,896 h. The median number of urgent items completed within the 2,281-hour threshold is 1,804 for Preemption, 1,799 for both Priority Jumping and Reserved Capacity, and 204 for Strict Expedite WIP Limit. Compared to the baseline, finish times rose by about 11% for Priority Jumping, Preemption, and Reserved Capacity, and by about 14% for Strict Expedite WIP Limit. The urgent items completed within the threshold stayed at 1,804 for Preemption, fell slightly for Priority Jumping and Reserved Capacity (from 1,802 to 1,799), and fell by about 20% from 255 to 204 for Strict Expedite WIP Limit.

4 Results

Policy	Metric	Median	SD	P5	P95
Priority Jumping	Finish time (h)	41,765	1,850	38,745	44,840
Preemption	Finish time (h)	41,836	1,850	38,787	44,872
Reserved Capacity	Finish time (h)	42,344	1,870	39,308	45,485
Strict Expedite WIP Limit	Finish time (h)	60,896	6,690	49,193	71,026
Priority Jumping	Urgent met threshold	1,799	10.3	1,782	1,816
Preemption	Urgent met threshold	1,804	10.0	1,787	1,820
Reserved Capacity	Urgent met threshold	1,799	10.5	1,781	1,816
Strict Expedite WIP Limit	Urgent met threshold	204	30.4	157	258

Table 4.9: 800-server: run-level variability of finish time and urgent items.

4.2.2 1,200 servers

Normal work

Table 4.10 reports the normal-work predictability metrics for the 1,200-server scenario. Strict Expedite WIP Limit produced the lowest percentile levels, while Reserved Capacity produced the highest. Priority Jumping and Preemption stayed close to each other, with differences below 45 h.

Policy	Metric	Median (h)	SD (h)	P5 (h)	P95 (h)
Priority Jumping	Normal P50	4,667	149	4,427	4,917
Preemption	Normal P50	4,651	148	4,417	4,898
Reserved Capacity	Normal P50	4,840	157	4,590	5,110
Strict Expedite WIP Limit	Normal P50	4,113	143	3,885	4,359
Priority Jumping	Normal P85	9,348	238	8,961	9,746
Preemption	Normal P85	9,383	236	9,001	9,778
Reserved Capacity	Normal P85	9,696	251	9,292	10,115
Strict Expedite WIP Limit	Normal P85	8,892	237	8,511	9,286
Priority Jumping	Normal P95	10,971	262	10,545	11,408
Preemption	Normal P95	11,011	261	10,585	11,445
Reserved Capacity	Normal P95	11,309	273	10,863	11,759
Strict Expedite WIP Limit	Normal P95	10,525	264	10,097	10,960
Priority Jumping	Spread	6,306	190	5,992	6,620
Preemption	Spread	6,359	193	6,048	6,684
Reserved Capacity	Spread	6,462	197	6,149	6,792
Strict Expedite WIP Limit	Spread	6,407	198	6,091	6,738

Table 4.10: 1,200-server: run-level variability of normal work.

The ordering by median spread is Priority Jumping (6,306 h) < Preemption (6,359 h) <

4 Results

Strict Expedite WIP Limit (6,407 h) < Reserved Capacity (6,462 h), and again does not match the ordering by percentile level. Standard deviations are small relative to the level of each metric, with a variation of about 3.1% to 3.5% at P50 and about 2.4% to 2.7% at P85 and P95. Compared to the baseline, all four policies showed lower normal-work percentile levels at 1,200 servers. The decrease was similar across policies. The median Normal P50 fell by about 24%, the median Normal P85 by about 22%, the median Normal P95 by about 19% to 20%, and the median spread by about 15% to 16%. The ordering of policies by percentile level and spread is unchanged.

Urgent work

Table 4.11 reports the urgent work metrics. Priority Jumping and Reserved Capacity are almost identical, and Strict Expedite WIP Limit reached the highest medians. The median urgent hit rate is 94.5% for Preemption, 94.4% for both Priority Jumping and Reserved Capacity, and 16.2% for Strict Expedite WIP Limit. Preemption events again occurred only under the Preemption policy, with a median count of 1,685 per run, down from the baseline value of 1,722. Standard deviations are small for the first three policies (variation between about 6% and 9% on the Lead Time percentiles, below 1% on the hit rate), while Strict Expedite WIP Limit shows a variation between about 9% and 13%.

Policy	Metric	Median	SD	P5	P95
Priority Jumping	Urgent P50 (h)	72.0	5.7	63.1	81.8
Preemption	Urgent P50 (h)	50.3	3.0	46.0	55.7
Reserved Capacity	Urgent P50 (h)	72.1	5.7	63.1	81.8
Strict Expedite WIP Limit	Urgent P50 (h)	11,100	1,400	9,020	13,600
Priority Jumping	Urgent P85 (h)	574	49.5	502	657
Preemption	Urgent P85 (h)	558	49.8	482	637
Reserved Capacity	Urgent P85 (h)	574	50.1	501	659
Strict Expedite WIP Limit	Urgent P85 (h)	21,300	2,100	18,100	25,000
Priority Jumping	Urgent P95 (h)	2,530	215	2,200	2,900
Preemption	Urgent P95 (h)	2,500	218	2,160	2,890
Reserved Capacity	Urgent P95 (h)	2,530	214	2,200	2,910
Strict Expedite WIP Limit	Urgent P95 (h)	24,200	2,270	20,800	28,200
Priority Jumping	Hit rate (%)	94.4	0.5	93.5	95.2
Preemption	Hit rate (%)	94.5	0.5	93.6	95.3
Reserved Capacity	Hit rate (%)	94.4	0.5	93.5	95.2
Strict Expedite WIP Limit	Hit rate (%)	16.2	2.0	13.0	19.5

Table 4.11: 1,200-server: run-level variability of urgent work.

Preemption once more stayed nearly the same. For Priority Jumping and Reserved Capacity, the median Urgent P50 fell from about 93.5 h to 72.0 h (about 23% lower),

4 Results

while their P85 fell by about 3% and their P95 stayed barely unchanged. Strict Expedite WIP Limit improved, with Lead Time percentiles down by about 19% to 21% and hit rate up from 13.4% to 16.2%. The policy ordering has not changed.

Run-level summary

Table 4.12 reports the run-level results. All four policies again completed the full workload of 21,402 items (1,910 urgent). Priority Jumping and Preemption finished close together (median finish times of 34,899 h and 34,974 h), Reserved Capacity finished about 415 h later (35,314 h), and Strict Expedite WIP Limit finished clearly last at 48,349 h. The median number of urgent items completed within the 2,281-hour threshold is 1,804 for Preemption, 1,803 for both Priority Jumping and Reserved Capacity, and 309 for Strict Expedite WIP Limit.

Policy	Metric	Median	SD	P5	P95
Priority Jumping	Finish time (h)	34,899	1,620	32,323	37,607
Preemption	Finish time (h)	34,974	1,610	32,419	37,641
Reserved Capacity	Finish time (h)	35,314	1,660	32,648	38,036
Strict Expedite WIP Limit	Finish time (h)	48,349	5,310	38,148	55,562
Priority Jumping	Urgent met threshold	1,803	10.1	1,786	1,819
Preemption	Urgent met threshold	1,804	10.1	1,787	1,820
Reserved Capacity	Urgent met threshold	1,803	10.0	1,786	1,819
Strict Expedite WIP Limit	Urgent met threshold	309	37.8	248	372

Table 4.12: 1,200-server: run-level variability of finish time and urgent items completed.

Compared to the baseline, finish times fell by about 7% for Priority Jumping, Preemption, and Reserved Capacity, and by about 9% for Strict Expedite WIP Limit. The urgent items completed within the threshold stayed at 1,804 for Preemption, rose slightly for Priority Jumping and Reserved Capacity (from 1,802 to 1,803), and rose from 255 to 309 for Strict Expedite WIP Limit, an increase of about 21%.

4.2.3 Summary across capacity levels

This section summarizes the robustness results across all three capacity levels (800, 1,000, and 1,200 servers). The baseline on 1,000 servers (Section 4.1) and the two robustness scenarios (Section 4.2.1 and 4.2.2) provide the specific details of the scenario. Figures 4.1 and 4.2 present the cross scenario view of the two main metrics. The remaining metrics are summarized in the text.

Capacity effect on normal-work predictability

Figure 4.1 compares the median normal-work P85 across capacity scenarios and policies. Normal P85 is used here because it is the central percentile-based predictability measure in this thesis. Not only does it reflect typical Lead Time behavior, but it also shows an elevated tail risk. It remains less extreme than the P95.

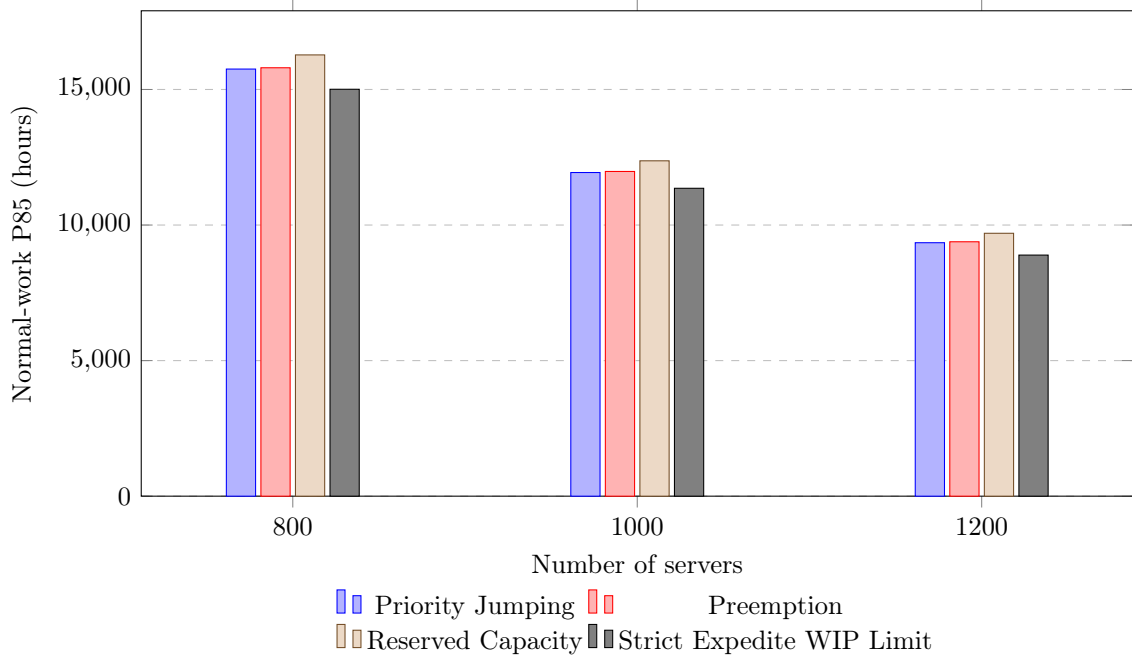


Figure 4.1: Median normal work P85 (hours) by policy across capacity scenarios.

The level of normal-work P85 decreases as capacity grows, and the decrease is similar across all four policies. At 800 servers, median P85 values range from 15,008 h (Strict Expedite WIP Limit) to 16,275 h (Reserved Capacity). At 1,000 servers, they range from 11,356 h to 12,369 h. At 1,200 servers, they range from 8,892 h to 9,696 h. The ordering of policies according to normal-work P85 is Strict Expedite WIP Limit < Priority Jumping < Preemption < Reserved Capacity, and this ordering is stable across all three capacity levels.

Capacity effect on urgent responsiveness

Figure 4.2 compares the median urgent hit rate across capacity scenarios and policies. The urgent hit rate is used here because it summarizes the urgent-work side of the trade-off in one value and makes the difference between Strict Expedite WIP Limit and the other three policies visible across all capacity levels.

4 Results

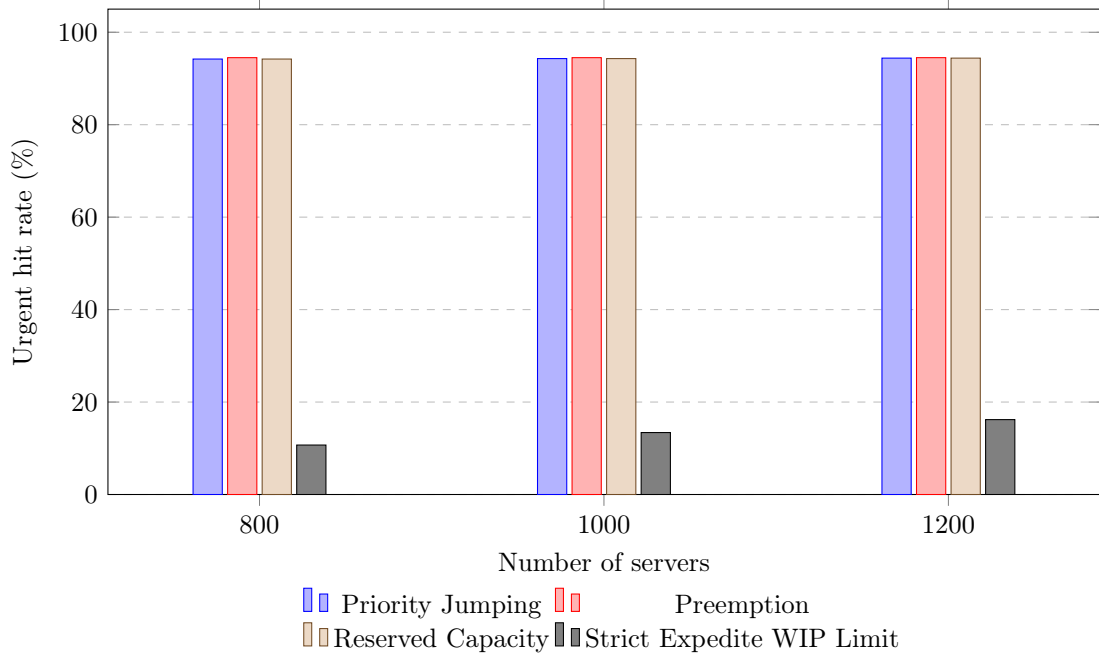


Figure 4.2: Median urgent hit rate (%) by policy across capacity scenarios.

For Priority Jumping, Preemption, and Reserved Capacity, the median urgent hit rate stays between 94.2% and 94.5% in all three capacity levels. The difference between these three policies, and between capacity levels within each of them, is at most 0.3 percentage points. Strict Expedite WIP Limit behaves differently. Its median urgent hit rate is 10.7% at 800 servers, 13.4% at 1,000 servers, and 16.2% at 1,200 servers. The difference between the Strict Expedite WIP Limit and the other three policies is the greatest at 800 servers and decreases as capacity increases, but it remains wide across all three scenarios.

Stability of policy ordering

The ordering of policies also holds for the metrics not shown in the two figures. For the spread measure, the ordering is Priority Jumping < Preemption < Strict Expedite WIP Limit < Reserved Capacity at all three capacity levels. This confirms the baseline pattern that spread and percentile levels rank the policies differently. For the urgent Lead Time percentiles, Preemption produces the shortest Lead Times at all three capacity levels, with a median Urgent P50 of 50.3 h in all three scenarios. Priority Jumping and Reserved Capacity stay close to each other at all capacity levels, with median Urgent P50 values of 149 h at 800 servers, 93 h at 1,000 servers, and 72 h at 1,200 servers. The median urgent P85 and P95 values of these two policies are also nearly identical across all scenarios. Strict Expedite WIP Limit remains clearly separated from the other three

4 Results

policies on all urgent Lead Time percentiles. For the run-level metrics, Priority Jumping and Preemption finished close together at every capacity level, Reserved Capacity finished later, and Strict Expedite WIP Limit finished clearly last. The completed workload of 21,402 items (1,910 urgent) is the same for all policies and capacity levels.

Sensitivity of Strict Expedite WIP Limit

Strict Expedite WIP Limit is the only policy whose urgent-work metrics change substantially with capacity. Its median urgent hit rate increases from 10.7% at 800 servers to 13.4% at 1,000 servers and 16.2% at 1,200 servers. The number of urgent items completed within the 2,281-hour threshold follows the same pattern, increasing from 204 at 800 servers to 255 at 1,000 servers and 309 at 1,200 servers. The urgent Lead Time percentiles move in the opposite direction. Median Urgent P50 decreases from 18,400 h at 800 servers to 13,963 h at 1,000 servers and 11,100 h at 1,200 servers, with Urgent P85 and P95 showing the same change. Strict Expedite WIP Limit improves on all urgent-work metrics as capacity grows. Even at 1,200 servers, it stays clearly below Priority Jumping, Preemption, and Reserved Capacity on urgent hit rate (16.2% versus 94.4% to 94.5%). The normal-work metrics of Strict Expedite WIP Limit follow the same capacity pattern as the other three policies and do not show this sensitivity.

4.3 Service time sensitivity

This section reports a service-time sensitivity analysis (Trim995) on the service-time input. The baseline scenario (Section 4.1) uses the full `work_hours` distribution without Trim995. To test whether the baseline findings depend on the most extreme right tail observations, the same scenario was re-run with the Trim995 service-time input, which caps values above the 99.5th percentile of the `work_hours` distribution. All other parameters remained unchanged. Each policy was executed again 10,000 times.

Policy	Metric	Baseline	Trim995	Difference
Priority Jumping	Finish time (h)	37,611	28,919	-8,692
Preemption	Finish time (h)	37,667	28,948	-8,719
Reserved Capacity	Finish time (h)	38,079	29,394	-8,685
Strict Expedite WIP Limit	Finish time (h)	53,233	43,158	-10,075
Priority Jumping	Urgent met threshold	1,802	1,802	0
Preemption	Urgent met threshold	1,804	1,804	0
Reserved Capacity	Urgent met threshold	1,802	1,802	0
Strict Expedite WIP Limit	Urgent met threshold	255	257	+2
Preemption	Total preemptions	1,722	1,722	0

Table 4.13: Comparison of median run-level metrics: baseline and trimmed.

4 Results

Tables 4.13 and 4.14 report the median values under the Trim995 input along with the baseline values. For normal work, the Trim995 medians differ from the baseline by at most 10 h on any percentile, which results in about 0.1%. For urgent work, the medians of Priority Jumping, Preemption, and Reserved Capacity differ from the baseline by at most 2 h on any percentile and below 0.1% on the urgent hit rate. Strict Expedite WIP Limit shows slightly larger differences on its urgent Lead Time percentiles (Urgent P85 falls from 26,369 h to 25,495 h, Urgent P95 from 29,879 h to 28,739 h), but its hit rate of 13.5% remains almost identical to the baseline value of 13.4%. The ordering of policies is unchanged on every reported metric.

Policy	Metric	Baseline	Trim995	Difference
Priority Jumping	Normal P50 (h)	6,143	6,157	+14
Preemption	Normal P50 (h)	6,130	6,132	+2
Reserved Capacity	Normal P50 (h)	6,388	6,387	-1
Strict Expedite WIP Limit	Normal P50 (h)	5,461	5,461	0
Priority Jumping	Normal P85 (h)	11,937	11,943	+6
Preemption	Normal P85 (h)	11,976	11,976	0
Reserved Capacity	Normal P85 (h)	12,369	12,359	-10
Strict Expedite WIP Limit	Normal P85 (h)	11,356	11,363	+7
Priority Jumping	Normal P95 (h)	13,594	13,603	+9
Preemption	Normal P95 (h)	13,646	13,649	+3
Reserved Capacity	Normal P95 (h)	14,067	14,059	-8
Strict Expedite WIP Limit	Normal P95 (h)	13,057	13,065	+8
Priority Jumping	Urgent P50 (h)	93.5	93.6	+0.1
Preemption	Urgent P50 (h)	50.3	50.4	+0.1
Reserved Capacity	Urgent P50 (h)	93.4	93.4	0
Strict Expedite WIP Limit	Urgent P50 (h)	13,963	13,957	-6
Priority Jumping	Urgent P85 (h)	592	593	+1
Preemption	Urgent P85 (h)	558	558	0
Reserved Capacity	Urgent P85 (h)	591	592	+1
Strict Expedite WIP Limit	Urgent P85 (h)	26,369	25,495	-874
Priority Jumping	Urgent P95 (h)	2,548	2,550	+2
Preemption	Urgent P95 (h)	2,510	2,510	0
Reserved Capacity	Urgent P95 (h)	2,549	2,549	0
Strict Expedite WIP Limit	Urgent P95 (h)	29,879	28,739	-1,140
Priority Jumping	Hit rate (%)	94.3	94.3	0.0
Preemption	Hit rate (%)	94.5	94.5	0.0
Reserved Capacity	Hit rate (%)	94.3	94.3	0.0
Strict Expedite WIP Limit	Hit rate (%)	13.4	13.5	+0.1

Table 4.14: Comparison of median normal work and urgent work metrics: baseline and trimmed.

4 Results

The only run-level metric that changes is the simulation finish time. Under the Trim995 input, finish times fall by about 23% for Priority Jumping, Preemption, and Reserved Capacity, and by about 19% for Strict Expedite WIP Limit. This shift reflects the removal of the most extreme service-time observations from the input distribution and is therefore expected. The order of the policies on the finish time is unchanged.

The baseline findings are robust to the existence of extreme right tail values in the service-time input. The four policies keep their behavior on normal-work predictability, urgent-work performance, and the urgent hit rate. Trimming only leads to a shorter completion time, and this occurs simply because extreme service-time values are removed, not because the underlying policy behaves differently.

5 Discussion

This chapter discusses the main findings of the empirical analysis and the simulation results. The goal is not to repeat the results, but to explain what they imply for the research questions and how they contribute to the overall argument of this thesis. The discussion first interprets the main data patterns and explains how the four urgent-work policies differ. It then connects these findings to the theoretical background on predictability, flow, urgent work, and classes of service. Finally, it evaluates the methodological strengths and limitations of the study and derives practical implications from the findings. The findings suggest that handling urgent work is a significant part of how a Kanban system operates. Each policy changes how delay is distributed between normal and urgent work. The differences are visible in normal-work Lead Time percentiles, in the spread, in urgent Lead Times, and in the share of urgent items completed within the urgent-work threshold. The main contribution of the analysis is not the identification of one best policy. Rather, the findings show that different urgent-work policies protect different parts of the system.

5.1 Interpretation of the main findings

The setup uses the SPARK project from the Public Jira Dataset and focuses on the 2015–2018 window. This window contains 210 weekly observations, with 19,492 normal and 1,910 urgent arrivals. The urgent-work threshold is derived from the urgent Lead Time distribution in the same period. The service-time input is based on `work_hours`, which is used as a proxy for active work time.

The Lead Time data already show high variability before the simulation is run. Both normal and urgent work have short median Lead Times compared to their upper percentiles. This supports the use of percentiles to evaluate predictability instead of relying only on averages. Averages would hide the long right tail of the distributions and would give an incomplete picture of delivery predictability. The data also show that the median Lead Times of urgent and normal work are close to each other (148.79 h and 158.88 h), while the urgent P85 is lower than the normal P85 (2,281 h versus 3,672 h). As a result, urgent items do not differ much from normal items in the typical case, but they have shorter upper tail Lead Times. This does not hold for the most extreme cases. At P99, urgent items have longer Lead Times than normal items (21,820 h vs 18,086 h). The urgent class has a shorter mid tail but a longer extreme tail. With 1,976 completed urgent items, the P99 depends on a small number of long running cases. This could be due to some stuck Blocker or Critical issues, or the fact that the Jira priority label does not always match

the true urgency, which is also discussed as a limitation in Section 5.5.

The policy comparison shows one clear exception and three other policies that perform very similarly on urgent work. Strict Expedite WIP Limit produced the lowest absolute normal-work percentile levels in the baseline scenario. Its normal-work P50, P85, and P95 were lower than under the other three policies. This came at a very high cost for urgent work. The urgent hit rate under Strict Expedite WIP Limit was much lower than under Priority Jumping, Preemption, and Reserved Capacity. Urgent Lead Times were also much higher. This suggests that the strict limitation protected normal work in terms of Lead Time duration, but shifted delay towards urgent items.

The poor performance of urgent work with Strict Expedite WIP Limit should not be interpreted as the result of a random small cap. The cap of 30 is already 30 times higher than the value of one that Vacanti (2025) uses as a reference. It was also set at the same level as the Reserved Capacity parameter so that the two policies can be compared on the same basis. A simple calculation explains why the cap of 30 still leads to long urgent Lead Times. Each urgent item takes about 558 hours of work on average (see Table 3.3). If at most 30 urgent items can be worked on at the same time, and if these 30 slots are continuously occupied, the maximum urgent completion rate is approximately $30/558 \approx 0.054$ urgent items per hour. Over one week, this corresponds to $0.054 \times 168 \approx 9.03$ urgent items. The SPARK data show an average of 9.10 urgent items arriving per week (see Table 3.2). The cap of 30 therefore allows the system to keep up with urgent arrivals only under average conditions, but it leaves almost no buffer for weeks in which urgent demand is above average. This also explains the robustness scenarios. With a cap of 24 (800 servers), the system can finish about 7 urgent items per week, which is below the arrival rate, so the queue grows. With a cap of 36 (1,200 servers), the system can finish about 11 urgent items per week, just above the arrival rate, but urgent arrivals are very uneven (their standard deviation is about as large as their mean, see Table 3.2), so this small buffer gets used up quickly. The pattern in the simulation matches what Vacanti (2025) describes for Strict Expedite WIP Limit in general, that the cap controls how often urgent work is allowed in, but it does not change the underlying problem.

Priority Jumping and Preemption formed a second pattern. Both policies stayed close to each other on normal-work metrics. Preemption had the strongest urgent-work performance, especially for urgent median Lead Time, while its normal-work percentiles stayed close to Priority Jumping. Under Preemption, the median Lead Time of urgent items is 50.3 hours at 800, 1,000, and 1,200 servers. This is the same as the median time it actually takes to do an urgent item, which is 50.35 hours (see Table 3.3). The reason is that with Preemption, an urgent item is started right away, so it does not spend any time waiting in the queue. Its Lead Time is equal to the time needed to do the work itself. The simulation reaches this lower limit, which is also a useful sign that the model behaves as expected. Also, under the chosen 24-hour preemption overhead, the interruption cost was not large enough to make normal-work predictability worse

5 Discussion

compared to non-preemptive priority. This result should not be generalized beyond the modeled overhead assumption. If the interruption cost were higher, Preemption could become less attractive. Reserved Capacity showed the weakest normal work results under the tested parameters. It produced the highest normal-work percentile levels and the highest spread consistently. At the same time, its urgent-work performance was almost identical to Priority Jumping. This result should not be interpreted as evidence that Reserved Capacity is generally ineffective. It shows that the 3% reserve used in this model had a permanent capacity cost on normal work, but was set too small to create a meaningful advantage for urgent work, as compared to non-preemptive priority. This interpretation is important because Reserved Capacity is structurally different from the other policies. It is the only non-work-conserving policy in the comparison, where capacity is withheld even when normal work is waiting. Setting a higher reserve could improve the response to urgent work, but it would also increase the amount of capacity that would be unavailable to normal work. In this case, the result points rather to a calibration issue than to the rejection of Reserved Capacity.

Reserved Capacity and Priority Jumping are not just close on urgent work, they produce nearly the same results. Their urgent percentiles differ by at most 1 h (P50 93.4 h vs 93.5 h, P85 591 h vs 592 h, P95 2,549 h vs 2,548 h), and their urgent hit rate is the same at 94.3%. The same applies at 800 and 1,200 servers. Reserved Capacity has the same urgent-work outcome as Priority Jumping, but at the cost of the highest normal-work percentile levels and the highest spread. Preemption helps urgent work mainly for the typical urgent item, not for the worst cases. Compared to Priority Jumping at baseline, Preemption cuts urgent P50 by about half (from 93.5 h to 50.3 h), but only reduces urgent P85 by about 6%, urgent P95 by about 1.5%, and the hit rate increases by 0.2%. If urgent performance is rated by a percentile-based threshold, the advantage of Preemption over Priority Jumping is much smaller than the urgent P50 numbers alone suggest.

Another finding is the difference between absolute percentile levels and distribution spread. Strict Expedite WIP Limit has the lowest normal-work P50, P85, and P95, but it does not have the lowest spread. Priority Jumping has the lowest spread, although its absolute percentile levels are higher. This difference matters because predictability is not only about lower Lead Times. It is also about the width of the Lead Time distribution. A policy can reduce the overall Lead Time level while still leaving a relatively wide tail.

The robustness scenarios support the stability of these findings. When capacity increases from 800 to 1,000 and 1,200 servers, normal-work Lead Time levels decrease in all policies, but the ordering of the policies remains stable. Strict Expedite WIP Limit has the lowest normal-work P85 at all capacity levels, followed by Priority Jumping, Preemption, and Reserved Capacity. For urgent work, Priority Jumping, Preemption, and Reserved Capacity stay close to each other, while Strict Expedite WIP Limit stays clearly below them. This suggests that the main policy differences are not only the result of the setting of baseline capacity. The service-time sensitivity analysis leads to the same conclusion.

Capping the largest 0.5% of service-time values reduces overall simulation finish times, but it does not change the policy ordering or the main interpretation. Normal-work and urgent-work percentiles change only slightly. This suggests that the comparison is not driven only by a small number of extreme service-time observations.

5.2 Answering the research questions

RQ1 asks how the Lead Time distribution of normal work differs from that of urgent work in a Kanban system based on real Jira data. The data show that urgent and normal work differ more strongly in the upper tail than in the median. The median Lead Time for urgent work is similar to that for normal work, but the urgent P85 and P95 values are lower than the matching values for normal work. This suggests that, in the SPARK data, grouping items by urgency does not greatly change the result for a typical item, but it is associated with a shorter upper tail of the distribution. At the same time, both classes show strong right skewness. This supports the use of percentile-based evaluation and shows that Lead Time predictability cannot be judged by the median or mean alone. This distribution pattern supports the use of urgent P85 as the performance threshold and the percentile-based evaluation framework used throughout the simulation.

RQ2 asks how different urgent-work policies affect predictability metrics for normal work. The simulation results show that the policies affect normal work in different ways. Strict Expedite WIP Limit produces the lowest normal-work percentiles, but not the lowest spread. Priority Jumping produces the lowest normal-work spread and therefore performs best when predictability is defined as having a tighter distribution. Preemption stays close to Priority Jumping on normal-work metrics, while Reserved Capacity performs the weakest for normal work. These findings show that policy choice changes the level and the shape of normal-work Lead Time distributions.

RQ3 asks what trade-offs occur between urgent-work performance and normal-work predictability under each policy. The strongest trade-off appears under Strict Expedite WIP Limit. It protects normal-work percentile levels, but urgent-work performance collapses compared to the other policies. Preemption produces the best urgent-work performance under the chosen overhead assumption and remains close to Priority Jumping on normal-work metrics. Priority Jumping appears as a balanced policy in this setup because it avoids interruption overhead and still provides strong urgent-work performance. Reserved Capacity creates a cost for normal work without a clear urgent-work advantage over Priority Jumping.

RQ4 asks under which capacity and service time conditions each policy performs best. Across the tested capacity levels, the policy ordering remains stable. Increasing capacity improves Lead Time levels, but does not change the ranking. The service time sensitivity check also does not change the interpretation. Therefore, within the tested

scenarios, the findings are robust to the capacity levels and service time trimming used in this thesis. This does not mean that the same ordering would hold for all possible parameter values. The Reserved Capacity share, the Strict Expedite WIP Limit, and the preemption overhead are design parameters. Different values could lead to different results.

5.3 Relation to theoretical background

The findings support the argument in theory that predictability should be assessed through the variation and percentiles in the Lead Time, not only through the mean values. The data and simulation outputs contain long right tails. This confirms the importance of the theoretical discussion on variability and averages. Averages alone would not show how strongly Lead Times differ between typical and extreme cases. Percentiles and spread provide a more useful view because they show both the central part and the tail of the distribution.

The results also support the view of Kanban as a flow system in which clearly defined policies are important. The four policies in this thesis are not only different priority rules. They are different ways of regulating sequencing and capacity allocation. Priority Jumping changes the order in the queue. Preemption changes both the order and the behavior of work interruption. Reserved Capacity separates part of the capacity for urgent work. Strict Expedite WIP Limit restricts how many urgent items may be active at the same time. These mechanisms lead to different outcomes, even though all policies process the same arrivals and service-time inputs.

The findings are also consistent with the theoretical idea that urgent work can disturb normal work. However, the results do not show a simple trade-off, where better urgent performance always leads to worse predictability for normal work. The pattern is more specific. Compared to Priority Jumping, Preemption improves urgent-work results without clearly worsening normal-work results, at least under the chosen overhead value. Reserved Capacity worsens normal work without improving urgent work much compared to Priority Jumping. Strict Expedite WIP Limit improves normal-work percentile levels but performs worse for urgent work. The main implication is therefore not that urgent work always harms normal work in the same way. The policy mechanism determines where the delay appears.

The findings also put into perspective the idea of having very strict WIP limits for expedite items. WIP limits are a central Kanban practice because they help control overload and improve flow. In this simulation, a strict urgent WIP cap has a negative effect on urgent-work performance. This does not imply that WIP limits are generally harmful. It rather indicates that an expedite WIP limit cannot be regarded as a standalone rule. It must be considered together with system capacity, urgent arrivals, and the purpose of the policy. In this setup, the chosen limit protects normal work but delays urgent items too

strongly.

The comparison between Priority Jumping and Preemption is also relevant for the theoretical discussion of interruptions. The theoretical background shows that interruptions and task switching can create costs. The simulation includes this through a 24-hour preemption overhead. Under this assumption, Preemption still performs well. It achieves the strongest urgent-work results and remains close to Priority Jumping for normal work. This indicates that preemption can be beneficial when the cost of handling interruptions is limited. At the same time, the result depends on the chosen overhead parameter. It should not be taken as proof that interruptions have no negative effects in real teams.

Reserved Capacity shows why the distinction between work-conserving and non-work-conserving policies matters. By reserving capacity for urgent work, the policy limits the capacity available for normal work. In the simulation, Reserved Capacity makes normal work outcomes worse, but does not clearly improve urgent work outcomes compared to Priority Jumping. Reserved Capacity may be more useful when urgent work appears more often, when urgent bursts need to be handled immediately, or when an organization wants to keep capacity available for urgent work at all times. Under the tested conditions, the 3 percent reserve did not create a clear advantage for urgent work over Priority Jumping. This suggests that Reserved Capacity is sensitive to calibration. A small reserve has the benefit of not disrupting the flow of normal work too much, but it may be too weak to handle frequent urgent arrivals. Vice versa, a large reserve could improve urgent responsiveness, but would also increase the capacity cost on normal work. As a consequence, the results suggest that the 3% setting should be seen as a conservative slack scenario, rather than as an optimal configuration for Reserved Capacity.

5.4 Methodological strengths

A main strength of the method is the combination of Jira data and controlled simulation. A purely empirical analysis of Jira data could show historical Lead Times, but it could not separate the effects of different urgent-work policies. The project followed its own real-world process, and the policy rules are not controlled. On the other hand, a purely theoretical simulation would make it possible to compare policies but would rely strongly on assumed distributions for arrival and service times. This thesis combines both approaches. Real Jira data are used to derive arrivals, Lead Time benchmarks, and service time samples, while the simulation creates controlled conditions to compare policies.

Another strength is the internal consistency of the setup. The arrivals, urgent-work threshold, and service-time input are all derived from the same SPARK 2015–2018 window. This avoids mixing data from different periods with potentially different workflow behavior. The window choice is also justified by the suitability of the data, especially the coverage of the derived `work_hours` variable.

5 Discussion

An advantage lies in the large number of replications. Each policy is executed 10,000 times per scenario. This supports stable run-level summaries, especially for metrics such as P85 and P95, which are more sensitive to tail behavior than means or medians. The results section also reports run-level variation, which makes it possible to see whether policy differences are stable across simulation runs. With 10,000 runs per policy, the median of each metric is known with very high precision. For Normal P50 under Priority Jumping, the standard deviation across runs is 202 h (see Table 4.2). Dividing by the square root of 10,000 gives a standard error of about 2 h, which is tiny compared to the metric itself. Even small differences between policies can be measured reliably. The question for the analysis is no longer whether the differences are real, but whether they are big enough to matter in practice. The 0.2% gap on urgent hit rate between Preemption and Priority Jumping at baseline is a good example. It can be detected reliably with this number of runs, but the effect is small, so any practical recommendations based on it should be made cautiously.

Furthermore, the policy constraints scale proportionally across different capacity scenarios. The Reserved Capacity share and the Strict Expedite WIP Limit are scaled with total capacity in the robustness scenarios. Without this, policy differences would shrink as capacity grows, and the comparison would lose its interpretive value. In addition, the thesis does not rely only on the 1000-server baseline. It also tests lower and higher capacity levels and repeats the baseline with a trimmed service-time input. This makes the interpretation more plausible because the main policy ordering remains stable across these checks. Finally, the assumptions of the model are transparent. The thesis clearly states that `work_hours` is a service time proxy and not a direct measurement of actual work time. It also states that the servers are abstract capacity units, not historical developers of the SPARK project. This transparency is important because it defines how the results should be interpreted.

5.5 Limitations

The first limitation concerns external validity. The analysis is based on an open-source Jira project. SPARK provides a large number of issues, priority information, and usable workflow history for the selected period. It is still only one project. Other software projects may have different urgency patterns, different team structures, different priority disciplines, and different meanings of the workflow state. The findings should therefore not be seen as simple, general rankings that apply to every Kanban system.

The second limitation concerns the Reserved Capacity parameter. The thesis tests Reserved Capacity with 3% slack. This value was chosen to protect normal work capacity while still keeping a small buffer for urgent arrivals. The study does not include a variation in the reserve parameter. So it cannot be determined whether a larger reserve would produce a stronger benefit for urgent work. The findings for Reserved Capa-

5 Discussion

city should be seen as evidence of the tested configuration, not of all possible reserve levels.

The third limitation concerns the proxy used for urgency. The study classifies Blocker and Critical issues as urgent and all remaining issues as normal. This is a practical and reproducible rule, but Jira priority is not the same as directly observed business urgency. Some urgent items may not be labeled Blocker or Critical, and some Blocker or Critical items may not have been treated as true expedite work in practice. The classification is therefore an operational approximation.

The fourth limitation concerns the service-time proxy. `work_hours` is derived from the time spent in specific workflow states, mainly active work states. It is not a direct measurement of actual effort. This matters because Jira workflow states may not always reflect real activity. An issue can remain in an active state while no one is working on it, or it can be worked on while the recorded status does not fully capture that effort. The simulation uses the best available proxy, but the absolute time values should not be read as exact work time.

The fifth limitation concerns the temporal resolution of arrivals and the load level of the simulated system. Arrivals are modeled on weekly resolution. This is suitable for the chosen setup and keeps the model manageable, but it does not reflect bursts within the week. This is important because urgent work may arrive in clusters. Such bursts could affect queue behavior, especially for policies that restrict urgent capacity or urgent WIP. A finer arrival model could produce additional insights. Another aspect to consider is the current load on the system. The total amount of work in the simulation is about 17.5 million hours, spread over 210 weeks. With 1,000 servers, this gives an average utilization of about 50%. With 800 servers, utilization is about 62%, and with 1,200 servers, about 41%. None of the scenarios is close to a fully loaded system. The policy ordering is therefore stable when the system is under moderate load, but not when it is under heavy load. Many real Kanban teams work at much higher utilization. Standard queueing theory predicts that policy differences grow as utilization approaches 100%. So the differences between the four policies may be larger in tighter systems than the ones tested here.

The sixth limitation concerns the simplified representation of the Kanban workflow in the simulation model. The servers in the model are abstract capacity units. They do not represent individual developers, team members, skills, calendars, or availability. The model also does not include dependencies between issues, rework, batching, handovers, or dynamic priority changes during processing. This simplified approach is necessary to compare policy mechanisms under controlled conditions, but it limits the realism of the simulation.

The seventh limitation concerns the interpretation of Preemption costs. At baseline, Preemption causes 1,722 preemption events per run, each adding 24 h of overhead. This

5 Discussion

adds up to about 41,000 extra hours of work per run. Because the model has 1,000 servers running in parallel, this extra work is shared widely. Per server, it adds only about 41 h, which matches the difference in the median finish time between Preemption (37,667 h) and Priority Jumping (37,610 h), which is 57 h. In a smaller team, the same number of interruptions could not be handled in parallel and would have a much greater impact. The limited impact of Preemption should be interpreted with caution. It is partly a result of the high level of parallel capacity in the model, and not only from the chosen 24-hour overhead value.

The eighth limitation concerns policy parameterization. The baseline uses 1,000 servers, 30 reserved urgent servers, a Strict Expedite WIP Limit of 30 urgent items, and a 24-hour preemption overhead. These values are part of the scenario design. Robustness scenarios vary capacity and adapt the relevant policy constraints accordingly, and sensitivity analysis trims service time outliers. The thesis does not systematically vary the reserved capacity share, the expedite WIP limit, or the preemption overhead. The findings therefore apply to the tested parameterization, not to all possible policy settings.

The final limitation concerns the absence of a priority-free FIFO or FCFS baseline. The comparison is between four urgent-work policies, not a comparison against a system without priority treatment. This means that the results show how the tested policies distribute delay between normal and urgent work relative to each other. They do not quantify the absolute amount of delay introduced by urgent-work handling compared to a neutral queue discipline. A FIFO baseline would make it possible to estimate this additional policy cost more directly. Since the focus of this thesis is the comparison of explicit urgent-work policies, such a baseline is left for future research.

5.6 Implications

The results suggest that teams should not choose an urgent-work policy only because it sounds aligned with Kanban principles. The policy must match the capacity of the system, the urgency of arrivals, and the performance goal. A strict expedite WIP limit can protect normal-work percentile levels, but in this model it performs very poorly for urgent work. This makes it difficult to treat it as a balanced policy under the tested conditions. Priority Jumping seems to be a simple and stable option in this setup. It gives urgent-work priority in the queue without interrupting Work in Progress and without reserving unused capacity. It also produces the lowest normal-work spread. This makes it attractive when a system wants to improve urgent responsiveness while keeping normal-work variability controlled.

Preemption appears useful when urgent responsiveness is the main concern and interruption overhead is limited. In the model, it produces the best urgent-work results and stays close to Priority Jumping on normal-work metrics. This depends on the chosen overhead

5 Discussion

value. In real teams, interruption costs may be higher, especially if the work requires deep context or specialized knowledge. Reserved Capacity requires careful calibration. The tested reserve does not create a clear urgent-work advantage over Priority Jumping, but it worsens normal-work metrics. Reserved Capacity can be valuable in environments where urgent requests arise frequently, where bursts must be served right away, or where an organization wants to ensure available response capacity regardless of the current state of the queue. Under the tested conditions, the chosen 3 percent reserve is not sufficient to produce a meaningful urgent-work benefit, while normal work pays a consistent capacity cost. The broader practical implication is that urgent-work policies should be evaluated as clearly defined policies. They should not remain informal decisions made under pressure. Kanban emphasizes clear policies because they make flow behavior observable and adjustable. The simulation results support this view. Once the policy is explicit, its effect on normal and urgent work can be compared.

6 Conclusion

Kanban systems are often used because teams need flow, visibility, and more reliable delivery expectations. These goals become harder to fulfill when urgent work enters the system, because urgent items compete with normal work for limited capacity and can change the order in which work is processed. Teams need clear policies for urgent work, but these policies can themselves change the predictability of the entire system. This thesis addressed this problem by combining Jira data from the SPARK project with a discrete-event simulation, in which four urgent-work policies were compared under controlled conditions: Priority Jumping, Preemption, Reserved Capacity, and Strict Expedite WIP Limit.

The central finding is that normal and urgent work differ less in their median Lead Times than in the upper tail of their distributions, and that the four urgent-work policies change how delay is distributed between the two classes. The comparison shows that Priority Jumping provided the most balanced outcome in this setup. Preemption produced the strongest urgent-work responsiveness. Strict Expedite WIP Limit reduced normal-work Lead Time levels at the expense of urgent work. Reserved Capacity shifted costs to normal work without a clear urgent-work advantage under the tested setting. These patterns remained stable across the tested capacity levels and the service-time sensitivity scenario. Together, the results support the theoretical view that predictability cannot be promised through averages alone. Both the empirical Lead Time data and the simulation output contain long right tails, which makes percentile-based metrics and the spread between P50 and P95 the more appropriate measures. In addition, distinguishing between work-conserving and non-work-conserving policies clarifies how each one truly affects the progression of work, beyond what its name is intended to suggest. In a literature where Kanban evidence is still largely descriptive and case-based, the contribution of this thesis is to show that explicit urgent-work policies can be compared empirically and that the differences between them are measurable. For research, the combination of Jira data and discrete-event simulation provides a reproducible way to evaluate urgent-work policies across different contexts. For practice, the results suggest that an urgent-work policy should be chosen based on what the team needs to protect, and that policy parameters such as a reserved capacity share or an expedite WIP limit need to be calibrated to the actual urgent arrival pattern rather than set by convention. Regarding tooling, this thesis shows that public Jira data can be used for simulation-based policy testing. This requires usable workflow histories and a clear service-time proxy, such as `work_hours`. The findings therefore apply within the boundaries of one Jira project, one calibration window, and a limited set of parameter settings for each policy. They should be read as comparative results from the SPARK 2015–2018 setup, not as

6 Conclusion

universal rankings for all Kanban systems. Future work could test the same policy comparison across additional Jira projects and vary the policy parameters more systematically.

This thesis shows that urgent-work handling changes predictability in Kanban-based workflows, but the effect depends on the policy mechanism. Under the tested conditions, no single policy was the best in every dimension, and the choice between them is a question of what the system aims to protect, rather than which policy sounds most aligned with Kanban principles. In practice, this implies that urgent-work policies should be defined explicitly rather than left to ad-hoc decisions. They should be designed, measured, and adjusted as explicit system policies because they shape how delay and predictability are distributed throughout the workflow.

6.1 Future research

Future research could extend this study in several directions. The model could be applied to additional Jira projects from the Public Jira Dataset. This would show whether the policy ordering observed in SPARK also appears in other projects with different issue volumes, priority patterns, and workflow behavior. The thesis tests a fixed Reserved Capacity share of 3 percent, which is conservative compared to the urgent arrival share of approximately 9 percent in the SPARK data. Future research could vary this share to identify whether a higher reserve produces a meaningful urgent-work advantage without worsening normal-work metrics. Future work could test a wider range of policy parameters. Reserved Capacity could be varied from smaller to larger shares of total capacity. Strict Expedite WIP Limit could be tested at different caps. This would show whether the poor urgent-work performance of Strict Expedite WIP Limit is mainly caused by the chosen limit or by the policy mechanism itself under the modeled conditions. The preemption overhead could be varied. The present thesis uses one overhead value. A broader sensitivity analysis could identify when Preemption starts to harm normal work predictability more strongly than Priority Jumping. This would make the practical interpretation of preemption clearer. Future research could use a finer arrival model. Daily or event level arrivals would make it possible to test the effect of urgent work bursts. This would be relevant for policies that limit urgent WIP or reserve urgent capacity. A FIFO baseline could also be included. This would allow for a direct comparison between urgent-work policies and a system without urgent prioritization. It would also make it possible to estimate how much delay is introduced by the handling of urgent work itself. Finally, future simulation models could include more organizational detail. Skills, calendars, team structures, dependencies, dynamic priority changes, or rework could be added. Such models would be more complex, but could show whether policy ordering remains stable when more realistic team behavior is included.

References

- Kanban University. (2022, December). The official guide to the kanban method (Version 2). https://kanban.university/wp-content/uploads/2023/04/The-Official-Kanban-Guide_A4.pdf
- Ahmad, M. O., Dennehy, D., Conboy, K., & Oivo, M. (2018). Kanban in software engineering: A systematic mapping study. *Journal of Systems and Software*, 137, 96–113. <https://doi.org/10.1016/j.jss.2017.11.045>
- Ozkan, N., Gürgeç Erdoğan, T., Bal, Ş., & Gök, M. Ş. (2022, September). Scrum, kanban or a mix of both? a systematic literature review. In M. Ganzha, L. A. Maciaszek, M. Paprzycki & D. Ślęzak (Eds.), *Proceedings of the 17th conference on computer science and intelligence systems (fedcsis)* (pp. 883–893, Vol. 30). PTI/IEEE. <https://doi.org/10.15439/2022F143>
- Open guide to kanban: In the context of knowledge work* [Adaptation of *The Kanban Guide* (May 2025) by Orderly Disruption Limited & Daniel S. Vacanti, Inc.; licensed under CC BY-SA 4.0]. (2025, July). Retrieved May 17, 2026, from <https://kanbanguides.org/open-guide-to-kanban/2025.7/>
- Vacanti, D. S. (2025, April). *Actionable agile metrics for predictability: Tenth anniversary edition* [Leanpub e-book; version published 2025-04-02]. ActionableAgile Press. <https://leanpub.com/aamfp-10th>
- Sjøberg, D. I. K., Johnsen, A., & Solberg, J. (2012). Quantifying the effect of using kanban versus scrum: A case study. *IEEE Software*, 29(5), 47–53. <https://doi.org/10.1109/MS.2012.110>
- Anderson, D., Concas, G., Lunesu, M. I., & Marchesi, M. (2011). Studying lean-kanban approach using software process simulation. In A. Sillitti, O. Hazzan, E. Bache & X. Albaladejo (Eds.), *Agile processes in software engineering and extreme programming: 12th international conference, xp 2011, madrid, spain, may 10–13, 2011, proceedings* (pp. 12–26, Vol. 77). Springer. https://doi.org/10.1007/978-3-642-20677-1_2
- Assunção, M. D., Cavalcante, V. F., Gatti, M. A. d. C., Netto, M. A. S., Pinhanez, C. S., & de Souza, C. R. B. (2012, December). Scheduling with preemption for incident management: When interrupting tasks is not such a bad idea. In C. Laroque, J. Himmelsbach, R. Pasupathy, O. Rose & A. M. Uhrmacher (Eds.), *Proceedings of the 2012 winter simulation conference (wsc)* (pp. 3675–3686). IEEE. <https://doi.org/10.1109/WSC.2012.6465101>
- Montgomery, L., Lüders, C., & Maalej, W. (2025, June). The public jira dataset [Version v7, published 23 June 2025]. <https://doi.org/10.5281/zenodo.15719919>

References

- Montgomery, L., Lüders, C., & Maalej, W. (2022). An alternative issue tracking dataset of public jira repositories. *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*, 73–77. <https://doi.org/10.1145/3524842.3528486>
- Petersen, K., & Wohlin, C. (2011). Measuring the flow in lean software development [The available PDF is a preprint dated 2009; this entry reflects the journal publication details.]. *Software: Practice and Experience*, 41(9), 975–996. <https://doi.org/10.1002/spe.975>
- Westin, A. (2016, October). *Managing software service operations with kanban* [Master's Thesis]. Aalto University, School of Science [Supervisor: Prof. Marjo Kauppinen; 78 pages].
- Harchol-Balter, M. (2013). *Performance modeling and design of computer systems: Queueing theory in action*. Cambridge University Press. <https://doi.org/10.1017/CBO9781139226424>
- Abdulahi, Y. (2023). *Optimization of telecommunications incident management process* [Master's Thesis]. Lappeenranta–Lahti University of Technology LUT, School of Engineering Science [Industrial Engineering and Management; Examiner: Prof. Janne Huiskonen; 87 pages].
- Li, Z., Cai, G., Yu, Q., Liang, P., Mo, R., & Liu, H. (2024). Bug priority change: An empirical study on apache projects. *Journal of Systems and Software*, 212, 112019. <https://doi.org/10.1016/j.jss.2024.112019>
- Shakeri Hossein Abad, Z., Karras, O., Schneider, K., Barker, K., & Bauer, M. (2018). Task interruption in software development projects: What makes some interruptions more disruptive than others? *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE '18)*, 122–132. <https://doi.org/10.1145/3210459.3210471>
- The AnyLogic Company. (2025, September). AnyLogic [Version 8.9.6, released 8 September 2025]. <https://www.anylogic.com>

A Appendix

A.0 Reproducibility: SPARK Jira Log Preparation

This appendix contains the scripts used to reproduce the thesis input datasets from the publicly available SPARK Jira data. The pipeline is written in R. Running `./run_repro.sh` starts the data preparation script (`repro_pipeline.R`) and then the validation script (`validate.R`). The validation step checks whether the generated outputs match the values used in this thesis.

The reproducibility pipeline follows the thesis baseline used in the main analysis. It restricts the thesis outputs to the 2015–2018 analysis window. Arrivals are based on `created_at`. Completion-based outputs use `done_at_final`, defined as `coalesce(resolutiondate, done_at_fallback)`. Service-time inputs are based on cleaned and capped `work_hours`.

Required input files:

- `SPARK_issues.csv`
- `SPARK_status_events.csv`

Prerequisites: R must be installed and `Rscript` must be available on the system `PATH`. No additional R packages are required.

A.1 README

```
# Reproducibility (SPARK Jira Log Preparation)

This folder reproduces the thesis input datasets for the SPARK
2015-2018 analysis window.

## Prerequisites
- R installed (Rscript available in terminal)
- Input files available in the dataset folder:
  - 'SPARK_issues.csv'
  - 'SPARK_status_events.csv'

## Run
Open a terminal, go to the 'repro/' folder, and run:

cd "/path/to/ThePublicJiraDataset_SPARK/repro"
```

```
./run_repro.sh
```

Listing A.1: README.txt – setup and usage

A.2 Entry point: run_repro.sh

The Bash script checks whether the input files exist, creates the output folder, and then runs the two R scripts in sequence.

```

1  #!/usr/bin/env bash
2  set -euo pipefail
3
4  DATA_DIR="${1:-$(cd "$(dirname "$0")/.." && pwd)}"
5  OUT_DIR="${2:-$DATA_DIR/repro_out}"
6
7  ISSUES="$DATA_DIR/SPARK_issues.csv"
8  EVENTS="$DATA_DIR/SPARK_status_events.csv"
9
10 if [[ ! -f "$ISSUES" ]]; then
11     echo "ERROR: Missing $ISSUES"
12     exit 1
13 fi
14
15 if [[ ! -f "$EVENTS" ]]; then
16     echo "ERROR: Missing $EVENTS"
17     exit 1
18 fi
19
20 mkdir -p "$OUT_DIR"
21
22 if ! command -v Rscript >/dev/null 2>&1; then
23     echo "ERROR: Rscript not found. Install R."
24     exit 1
25 fi
26
27 Rscript --vanilla "$(dirname "$0")/repro_pipeline.R" "$DATA_DIR" "
28     $OUT_DIR"
29 Rscript --vanilla "$(dirname "$0")/validate.R" "$OUT_DIR"
30 echo "DONE. Outputs in: $OUT_DIR"
```

Listing A.2: run_repro.sh – entry script

A.3 Data preparation pipeline: repro_pipeline.R

This script reads the raw SPARK CSV files, parses timestamps, reconstructs completion timestamps, classifies issues as *normal* or *urgent*, builds the 2015–2018 weekly arrival series, computes Lead Times, reconstructs status stays, and exports the cleaned and capped `work_hours` service-time samples used in the simulation.

A Appendix

```

1 args <- commandArgs(trailingOnly = TRUE)
2 if (length(args) < 2) stop("Usage: Rscript repro_pipeline.R <DATA_DIR> <
   OUT_DIR>")
3
4 DATA_DIR <- args[1]
5 OUT_DIR <- args[2]
6
7 issues_path <- file.path(DATA_DIR, "SPARK_issues.csv")
8 events_path <- file.path(DATA_DIR, "SPARK_status_events.csv")
9
10 issues <- read.csv(issues_path, stringsAsFactors = FALSE)
11 events <- read.csv(events_path, stringsAsFactors = FALSE)
12
13 parse_utc <- function(x) {
14   x <- as.character(x)
15   x[x == ""] <- NA
16   x <- sub("^00([0-9]{2})-", "20\\1-", x)
17
18   fmts <- c(
19     "%Y-%m-%dT%H:%M:%OS%z",
20     "%Y-%m-%dT%H:%M:%OS",
21     "%Y-%m-%d %H:%M:%OS%z",
22     "%Y-%m-%d %H:%M:%OS"
23   )
24
25   out <- rep(as.POSIXct(NA), length(x))
26   for (f in fmts) {
27     miss <- is.na(out) & !is.na(x)
28     if (!any(miss)) break
29     parsed <- as.POSIXct(strptime(x[miss], format = f, tz = "UTC"), tz =
       "UTC")
30     out[miss] <- parsed
31   }
32   out
33 }
34
35 issues$created_at <- parse_utc(issues$created_at)
36 issues$resolutiondate <- parse_utc(issues$resolutiondate)
37 events$transition_at <- parse_utc(events$transition_at)
38
39 DONE_STATUSES <- c("Resolved", "Closed")
40 URGENT_PRIOS <- c("Blocker", "Critical")
41
42 WINDOW_START <- as.POSIXct("2015-01-01 00:00:00", tz = "UTC")
43 WINDOW_END <- as.POSIXct("2019-01-01 00:00:00", tz = "UTC") #
   exclusive
44
45 week_start_monday <- function(ts) {
46   d <- as.Date(ts, tz = "UTC")
47   w <- as.integer(strftime(d, "%u")) # 1 = Monday, ..., 7 = Sunday
48   d - (w - 1)
49 }
50

```

A Appendix

```

51 # -----
52 # 1) Reconstruct completion fallback
53 # -----
54 events_done <- events[events$to_status %in% DONE_STATUSES & !is.na(events
    $transition_at), ]
55 done_fallback <- aggregate(transition_at ~ issue_key, data = events_done,
    FUN = max)
56 names(done_fallback)[2] <- "done_at_fallback"
57
58 write.csv(done_fallback,
59           file.path(OUT_DIR, "SPARK_done_fallback_repro.csv"),
60           row.names = FALSE)
61
62 # -----
63 # 2) Attach fallback and classify issues
64 # -----
65 issues2 <- merge(issues, done_fallback, by = "issue_key", all.x = TRUE)
66
67 issues2$done_at_final <- issues2$resolutiondate
68 miss <- is.na(issues2$done_at_final)
69 issues2$done_at_final[miss] <- issues2$done_at_fallback[miss]
70
71 issues2$class <- ifelse(!is.na(issues2$priority) & issues2$priority %in%
    URGENT_PRIOS,
72                    "urgent", "normal")
73
74 # -----
75 # 3) Arrivals in the 2015-2018 window
76 # -----
77 arr_src <- issues2[
78   !is.na(issues2$created_at) &
79   issues2$created_at >= WINDOW_START &
80   issues2$created_at < WINDOW_END,
81   c("issue_key", "created_at", "class")
82 ]
83
84 arr_src$week <- week_start_monday(arr_src$created_at)
85
86 arr <- aggregate(issue_key ~ week + class, data = arr_src, FUN = length)
87 names(arr)[3] <- "n"
88
89 full_weeks <- seq(as.Date("2014-12-29"), as.Date("2018-12-31"), by = "7
    days")
90
91 full_grid <- data.frame(
92   week = rep(full_weeks, each = 2),
93   class = rep(c("normal", "urgent"), times = length(full_weeks)),
94   stringsAsFactors = FALSE
95 )
96
97 arr_full <- merge(full_grid, arr, by = c("week", "class"), all.x = TRUE)
98 arr_full$n[is.na(arr_full$n)] <- 0L
99 arr_full <- arr_full[order(arr_full$week, arr_full$class), ]

```

A Appendix

```
100 write.csv(arr_full,
101           file.path(OUT_DIR, "SPARK_arrivals_weekly_2015_2018_long_repro.
102                        csv"),
103           row.names = FALSE)
104
105 arr_wide <- data.frame(
106   week = arr_full$week[arr_full$class == "normal"],
107   normal = arr_full$n[arr_full$class == "normal"],
108   urgent = arr_full$n[arr_full$class == "urgent"]
109 )
110
111 write.csv(arr_wide,
112           file.path(OUT_DIR, "SPARK_arrivals_weekly_2015_2018_wide_repro.
113                        csv"),
114           row.names = FALSE)
115
116 # -----
117 # 4) Completed issues in the 2015-2018 window
118 # -----
119 completed_window <- issues2[
120   !is.na(issues2$created_at) &
121   !is.na(issues2$done_at_final) &
122   issues2$done_at_final >= WINDOW_START &
123   issues2$done_at_final < WINDOW_END,
124 ]
125
126 completed_window$lead_time_hours <- as.numeric(
127   difftime(completed_window$done_at_final,
128             completed_window$created_at,
129             units = "hours")
130 )
131
132 metrics_out <- completed_window[, c(
133   "issue_key", "project_key", "created_at", "done_at_final",
134   "issuetype", "priority", "class", "lead_time_hours"
135 )]
136
137 write.csv(metrics_out,
138           file.path(OUT_DIR, "SPARK_issues_metrics_class_2015_2018_repro.
139                        csv"),
140           row.names = FALSE)
141
142 # -----
143 # 5) Reconstruct status stays for completed issues in window
144 # -----
145 events2 <- events[
146   events$issue_key %in% completed_window$issue_key &
147   !is.na(events$transition_at),
148 ]
149 events2 <- events2[order(events2$issue_key, events2$transition_at), ]
150
151 build_stays_one <- function(issue_key, ev, created_at, done_at) {
```

A Appendix

```

150 if (nrow(ev) == 0) {
151   return(data.frame(
152     issue_key = issue_key,
153     status = "NoStatusHistory",
154     entered_at = created_at,
155     left_at = done_at,
156     duration_hours = as.numeric(difftime(done_at, created_at, units = "
157       hours")),
158     stringsAsFactors = FALSE
159   ))
160 }
161
162 ev <- ev[order(ev$transition_at), ]
163 stays <- vector("list", nrow(ev) + 1)
164
165 stays[[1]] <- data.frame(
166   issue_key = issue_key,
167   status = ev$from_status[1],
168   entered_at = created_at,
169   left_at = ev$transition_at[1],
170   duration_hours = as.numeric(difftime(ev$transition_at[1], created_at,
171     units = "hours")),
172   stringsAsFactors = FALSE
173 )
174
175 if (nrow(ev) >= 2) {
176   for (i in 2:nrow(ev)) {
177     stays[[i]] <- data.frame(
178       issue_key = issue_key,
179       status = ev$from_status[i],
180       entered_at = ev$transition_at[i - 1],
181       left_at = ev$transition_at[i],
182       duration_hours = as.numeric(
183         difftime(ev$transition_at[i], ev$transition_at[i - 1], units =
184           "hours")
185       ),
186       stringsAsFactors = FALSE
187     )
188   }
189 }
190
191 stays[[nrow(ev) + 1]] <- data.frame(
192   issue_key = issue_key,
193   status = ev$to_status[nrow(ev)],
194   entered_at = ev$transition_at[nrow(ev)],
195   left_at = done_at,
196   duration_hours = as.numeric(difftime(done_at, ev$transition_at[nrow(
197     ev)], units = "hours")),
198   stringsAsFactors = FALSE
199 )
200
201 do.call(rbind, stays)
202 }

```

A Appendix

```

199 comp_keys      <- completed_window$issue_key
200 comp_created   <- completed_window$created_at; names(comp_created) <- comp_
201 keys
202 comp_done      <- completed_window$done_at_final; names(comp_done) <- comp_
203 keys
204 split_events   <- split(events2, events2$issue_key)
205
206 all_stays <- lapply(comp_keys, function(k) {
207   ev <- split_events[[k]]
208   if (is.null(ev)) ev <- events2[0, ]
209   build_stays_one(k, ev, comp_created[[k]], comp_done[[k]])
210 })
211
212 stays_df <- do.call(rbind, all_stays)
213 stays_df <- merge(stays_df, completed_window[, c("issue_key", "class")],
214   by = "issue_key", all.x = TRUE)
215 stays_df <- stays_df[order(stays_df$issue_key, stays_df$entered_at), ]
216
217 write.csv(stays_df,
218   file.path(OUT_DIR, "SPARK_status_stays_class_2015_2018_repro.
219   csv"),
220   row.names = FALSE)
221
222 # -----
223 # 6) Build service-time samples from work_hours
224 # -----
225 norm_status <- function(s) tolower(trimws(as.character(s)))
226 stays_df$status_norm <- norm_status(stays_df$status)
227
228 sum_hours <- function(df, name_exact) {
229   sum(df$duration_hours[df$status_norm == name_exact], na.rm = TRUE)
230 }
231
232 service_by_issue <- lapply(split(stays_df, stays_df$issue_key), function(
233   df) {
234     open_h <- sum_hours(df, "open")
235     ip_h   <- sum_hours(df, "in progress")
236     rep_h  <- sum_hours(df, "reopened")
237     ns_h   <- sum_hours(df, "nostatushistory")
238
239     total_h <- sum(df$duration_hours, na.rm = TRUE)
240     work_h  <- ip_h + rep_h
241     wait_h  <- open_h
242
243     data.frame(
244       issue_key      = df$issue_key[1],
245       class          = df$class[1],
246       open_hours     = open_h,
247       in_progress_hours = ip_h,
248       reopened_hours  = rep_h,
249       nostatus_hours  = ns_h,

```

A Appendix

```
248     total_hours      = total_h,
249     work_hours       = work_h,
250     wait_hours       = wait_h,
251     stringsAsFactors = FALSE
252   )
253 })
254
255 service_df <- do.call(rbind, service_by_issue)
256
257 write.csv(service_df,
258           file.path(OUT_DIR, "SPARK_service_times_by_issue_2015_2018_
259                        repro.csv"),
260           row.names = FALSE)
261
262 svc_emp <- service_df[service_df$work_hours > 0, c("issue_key", "class")]
263 svc_emp$service_hours <- service_df$work_hours[service_df$work_hours > 0]
264
265 write.csv(svc_emp,
266           file.path(OUT_DIR, "SPARK_service_workhours_2015_2018_empirical
267                        _repro.csv"),
268           row.names = FALSE)
269
270 cap_by_class <- tapply(svc_emp$service_hours,
271                       svc_emp$class,
272                       quantile,
273                       probs = 0.995,
274                       na.rm = TRUE)
275
276 svc_trim <- svc_emp
277 svc_trim$service_hours <- mapply(
278   function(h, cl) min(h, cap_by_class[[cl]]),
279   svc_trim$service_hours,
280   svc_trim$class
281 )
282
283 write.csv(svc_trim,
284           file.path(OUT_DIR, "SPARK_service_workhours_2015_2018_clean_
285                        trim995_repro.csv"),
286           row.names = FALSE)
```

Listing A.3: repro_pipeline.R – data preparation pipeline

A.4 Validation script: validate.R

After the pipeline finishes, the validation script checks whether the generated outputs match the values used in the thesis. If a check fails, the script stops with a non-zero exit code and prints a **VALIDATION: FAIL** message. If all checks pass, the script prints **VALIDATION: PASS** together with a short summary.

```
1 args <- commandArgs(trailingOnly = TRUE)
2 if (length(args) < 1) stop("Usage: Rscript validate.R <OUT_DIR>")
```

A Appendix

```

3 OUT_DIR <- args[1]
4
5 fail <- function(msg) { cat("VALIDATION: FAIL -", msg, "\n"); quit(status
6   = 1) }
7
8 ok <- function(msg) { cat(" -", msg, "\n") }
9
10 metrics <- read.csv(
11   file.path(OUT_DIR, "SPARK_issues_metrics_class_2015_2018_repro.csv"),
12   stringsAsFactors = FALSE
13 )
14
15 arr_w <- read.csv(
16   file.path(OUT_DIR, "SPARK_arrivals_weekly_2015_2018_wide_repro.csv"),
17   stringsAsFactors = FALSE
18 )
19
20 svc <- read.csv(
21   file.path(OUT_DIR, "SPARK_service_workhours_2015_2018_empirical_repro.
22     csv"),
23   stringsAsFactors = FALSE
24 )
25
26 svc_trim <- read.csv(
27   file.path(OUT_DIR, "SPARK_service_workhours_2015_2018_clean_trim995_
28     repro.csv"),
29   stringsAsFactors = FALSE
30 )
31
32 # -----
33 # Expected reference values
34 # -----
35
36 EXP_completed_total <- 18826
37 EXP_completed_normal <- 16850
38 EXP_completed_urgent <- 1976
39
40 EXP_arr_weeks <- 210
41 EXP_arr_week_min <- "2014-12-29"
42 EXP_arr_week_max <- "2018-12-31"
43
44 EXP_arr_total_normal <- 19492
45 EXP_arr_total_urgent <- 1910
46 EXP_arr_total_all <- 21402
47 EXP_arr_mean_normal <- 92.81904761904762
48 EXP_arr_mean_urgent <- 9.095238095238095
49
50
51 EXP_lt_urgent_n <- 1976
52 EXP_lt_urgent_mean <- 1479.62
53 EXP_lt_urgent_p50 <- 148.79
54 EXP_lt_urgent_p85 <- 2281.06
55 EXP_lt_urgent_p95 <- 8262.92
56 EXP_lt_urgent_p99 <- 21820.13
57
58
59 EXP_service_rows <- 12902
60 EXP_service_normal <- 11609

```

A Appendix

```
53 EXP_service_urgent <- 1293
54 EXP_cap_normal <- 15280.943986
55 EXP_cap_urgent <- 15246.597796
56
57 tol_small <- 1e-6
58 tol_round <- 0.05
59
60 # -----
61 # 1) Completed issue counts
62 # -----
63 ct_total <- nrow(metrics)
64 ct_norm <- sum(metrics$class == "normal")
65 ct_urg <- sum(metrics$class == "urgent")
66
67 if (ct_total != EXP_completed_total)
68   fail(paste("completed total", ct_total, "!=" , EXP_completed_total))
69   else ok("completed total OK")
70 if (ct_norm != EXP_completed_normal)
71   fail(paste("completed normal", ct_norm, "!=" , EXP_completed_normal))
72   else ok("completed normal OK")
73 if (ct_urg != EXP_completed_urgent)
74   fail(paste("completed urgent", ct_urg, "!=" , EXP_completed_urgent))
75   else ok("completed urgent OK")
76
77 # -----
78 # 2) Arrival series checks
79 # -----
80 if (nrow(arr_w) != EXP_arr_weeks)
81   fail(paste("arrival weeks", nrow(arr_w), "!=" , EXP_arr_weeks)) else ok(
82     "arrival weeks OK")
83
84 week_min <- as.character(min(as.Date(arr_w$week)))
85 week_max <- as.character(max(as.Date(arr_w$week)))
86
87 if (week_min != EXP_arr_week_min)
88   fail(paste("arrival min week", week_min, "!=" , EXP_arr_week_min)) else
89   ok("arrival min week OK")
90 if (week_max != EXP_arr_week_max)
91   fail(paste("arrival max week", week_max, "!=" , EXP_arr_week_max)) else
92   ok("arrival max week OK")
93
94 arr_tot_norm <- sum(arr_w$normal)
95 arr_tot_urg <- sum(arr_w$urgent)
96 arr_tot_all <- arr_tot_norm + arr_tot_urg
97
98 if (arr_tot_norm != EXP_arr_total_normal)
99   fail(paste("arrival total normal", arr_tot_norm, "!=" , EXP_arr_total_
100 normal)) else ok("arrival total normal OK")
101 if (arr_tot_urg != EXP_arr_total_urgent)
102   fail(paste("arrival total urgent", arr_tot_urg, "!=" , EXP_arr_total_
103 urgent)) else ok("arrival total urgent OK")
104 if (arr_tot_all != EXP_arr_total_all)
105   fail(paste("arrival total all", arr_tot_all, "!=" , EXP_arr_total_all))
```

A Appendix

```

    else ok("arrival total all OK")
98
99 arr_mean_norm <- mean(arr_w$normal)
100 arr_mean_urg <- mean(arr_w$urgent)
101
102 if (abs(arr_mean_norm - EXP_arr_mean_normal) > tol_small)
103   fail(paste("arrival mean normal", arr_mean_norm, "!=", EXP_arr_mean_
    normal)) else ok("arrival mean normal OK")
104 if (abs(arr_mean_urg - EXP_arr_mean_urgent) > tol_small)
105   fail(paste("arrival mean urgent", arr_mean_urg, "!=", EXP_arr_mean_
    urgent)) else ok("arrival mean urgent OK")
106
107 # -----
108 # 3) Urgent lead-time validation
109 # -----
110 urgent_lt <- metrics$lead_time_hours[metrics$class == "urgent"]
111
112 if (length(urgent_lt) != EXP_lt_urgent_n)
113   fail(paste("urgent lead-time n", length(urgent_lt), "!=", EXP_lt_urgent
    _n)) else ok("urgent lead-time n OK")
114
115 check_close <- function(value, expected, label) {
116   if (abs(value - expected) > tol_round) {
117     fail(paste(label, value, "!=", expected))
118   } else {
119     ok(paste(label, "OK"))
120   }
121 }
122
123 check_close(mean(urgent_lt), EXP_lt_urgent_mean, "urgent lead-time mean")
124 check_close(as.numeric(quantile(urgent_lt, 0.50, na.rm = TRUE)), EXP_lt_
    urgent_p50, "urgent lead-time p50")
125 check_close(as.numeric(quantile(urgent_lt, 0.85, na.rm = TRUE)), EXP_lt_
    urgent_p85, "urgent lead-time p85")
126 check_close(as.numeric(quantile(urgent_lt, 0.95, na.rm = TRUE)), EXP_lt_
    urgent_p95, "urgent lead-time p95")
127 check_close(as.numeric(quantile(urgent_lt, 0.99, na.rm = TRUE)), EXP_lt_
    urgent_p99, "urgent lead-time p99")
128
129 # -----
130 # 4) Service sample checks
131 # -----
132 svc_rows <- nrow(svc)
133 svc_norm <- sum(svc$class == "normal")
134 svc_urg <- sum(svc$class == "urgent")
135
136 if (svc_rows != EXP_service_rows)
137   fail(paste("service rows", svc_rows, "!=", EXP_service_rows)) else ok("
    service rows OK")
138 if (svc_norm != EXP_service_normal)
139   fail(paste("service normal rows", svc_norm, "!=", EXP_service_normal))
    else ok("service normal rows OK")
140 if (svc_urg != EXP_service_urgent)

```

A Appendix

```

141   fail(paste("service urgent rows", svc_urg, "!=", EXP_service_urgent))
142       else ok("service urgent rows OK")
143 trim_cap_normal <- max(svc_trim$service_hours[svc_trim$class == "normal"
144   ])
145 trim_cap_urgent <- max(svc_trim$service_hours[svc_trim$class == "urgent"
146   ])
147 if (abs(trim_cap_normal - EXP_cap_normal) > tol_small)
148   fail(paste("trim cap normal", trim_cap_normal, "!=", EXP_cap_normal))
149   else ok("trim cap normal OK")
150 if (abs(trim_cap_urgent - EXP_cap_urgent) > tol_small)
151   fail(paste("trim cap urgent", trim_cap_urgent, "!=", EXP_cap_urgent))
152   else ok("trim cap urgent OK")
153 cat("VALIDATION: PASS\n")
154 # -----
155 # Summary output
156 # -----
157 fmt <- function(x, d = 3) format(round(x, d), nsmall = d, scientific =
158   FALSE)
159 summ_vec <- function(x) {
160   x <- x[is.finite(x)]
161   c(
162     n      = length(x),
163     mean   = mean(x),
164     sd     = sd(x),
165     p50    = as.numeric(quantile(x, 0.50, na.rm = TRUE)),
166     p85    = as.numeric(quantile(x, 0.85, na.rm = TRUE)),
167     p95    = as.numeric(quantile(x, 0.95, na.rm = TRUE)),
168     p99    = as.numeric(quantile(x, 0.99, na.rm = TRUE)),
169     max    = max(x)
170   )
171 }
172 print_stats <- function(title, x_by_class) {
173   cat("\n", title, "\n", sep = "")
174   for (cl in names(x_by_class)) {
175     s <- summ_vec(x_by_class[[cl]])
176     cat(" - ", cl,
177       ": n=", s["n"],
178       ", mean=", fmt(s["mean"]),
179       ", sd=", fmt(s["sd"]),
180       ", p50=", fmt(s["p50"]),
181       ", p85=", fmt(s["p85"]),
182       ", p95=", fmt(s["p95"]),
183       ", p99=", fmt(s["p99"]),
184       ", max=", fmt(s["max"]),
185       "\n", sep = "")
186   }
187 }

```

A Appendix

```
188
189 cat("\n=== METRICS SUMMARY (from repro_out) ===\n")
190
191 cat("\nCompleted issues in validation window:\n")
192 cat(" - total: ", ct_total, "\n", sep = "")
193 cat(" - normal: ", ct_norm, "\n", sep = "")
194 cat(" - urgent: ", ct_urg, "\n", sep = "")
195
196 cat("\nArrivals in analysis window:\n")
197 cat(" - weeks: ", nrow(arr_w), " (", week_min, " .. ", week_max, ") \n",
    sep = "")
198 cat(" - totals: normal=", arr_tot_norm, ", urgent=", arr_tot_urg, ", all=",
    ", arr_tot_all", "\n", sep = "")
199 cat(" - mean/week: normal=", fmt(arr_mean_norm, 6),
200     ", urgent=", fmt(arr_mean_urg, 6),
201     ", all=", fmt(mean(arr_w$normal + arr_w$urgent), 6), "\n", sep = "")
202 cat(" - sd/week: normal=", fmt(sd(arr_w$normal)),
203     ", urgent=", fmt(sd(arr_w$urgent)),
204     ", all=", fmt(sd(arr_w$normal + arr_w$urgent)), "\n", sep = "")
205
206 lt_norm <- metrics$lead_time_hours[metrics$class == "normal"]
207 lt_urg <- metrics$lead_time_hours[metrics$class == "urgent"]
208
209 print_stats("Lead time (hours) by class:",
210            list(normal = lt_norm, urgent = lt_urg))
211
212 svc_norm_vec <- svc$service_hours[svc$class == "normal"]
213 svc_urg_vec <- svc$service_hours[svc$class == "urgent"]
214
215 print_stats("Service time input (clean work_hours) by class:",
216            list(normal = svc_norm_vec, urgent = svc_urg_vec))
217
218 cat("\n=== END SUMMARY ===\n")
```

Listing A.4: validate.R – validation and summary output

A.5 Simulation model documentation

The simulation model was implemented in AnyLogic. The following pages contain the model documentation exported from AnyLogic. This documentation reports the implemented model structure, parameters, functions, events, input files, and experiment settings used in this thesis.

The exported documentation is included for transparency and reproducibility. It complements the description in Section 3 by showing the implemented model configuration in the simulation software.

A Appendix

KanbanExpeditePolicies

Model: KanbanExpeditePolicies

Name	Value
General	
Model time units	hours
System Dynamics solver	
Differentiation Equations Method	Euler
Algebraic Equations Method	Modified Newton
Mixed Equations Method	RK45+Newton
Absolute accuracy	1.0E-5
Time accuracy	1.0E-5
Relative accuracy	1.0E-5
Fixed time step	0.001
Advanced	
Java package name	kanbanexpeditepolicies
File Name	/Users/ /Models/KanbanExpeditePolicies/KanbanExpeditePolicies.alp

Agent Type: Main

Name	Value
Agent actions	
Startup code	<pre>runId = 100 * (((com.anylogic.engine.ExperimentParamVariation)getExperiment()).getCu rrentIteration() - 1) + ((com.anylogic.engine.ExperimentParamVariation)getExperiment()).getCu rrentReplication(); currentSeed = inputSeed; if(policyType == 1) { activePolicyLabel = "Priority Jumping"; } else if(policyType == 2) { activePolicyLabel = "Preemption"; } else if(policyType == 3) { activePolicyLabel = "Reserved Capacity"; } else if(policyType == 4) { activePolicyLabel = "Strict Expedite WIP Limit"; } else { activePolicyLabel = "Unknown"; } for(int i = 0; i < servers.size(); i++) { servers.get(i).reservedForUrgent = (i < reservedUrgentServers); } loadServiceSamples(); writeResultsHeaderIfNeeded(); if(useToyData) { injectToyData(); } else {</pre>
Agent in flowcharts	
Use in flowcharts as	Agent
Dimensions and movement	

A Appendix

KanbanExpeditePolicies

Name	Value
Speed	(10 : MPS)
Rotate animation towards movement	true
Rotate vertically (incline on ramps)	false
Space and network	
Space Type	Continuous
Layout Type	User-defined
Network type	User-defined
Enable steps	false
Advanced Java	
Generic	false
Omniverse	
Usd binding type	None
Advanced	
Flowchart block	false
Logging	true
Auto-create datasets	true
AOC_DATASETS_UPDATE_TIME_PROPERTIES	- Recurring Event Properties
Limit the number of data samples	false



Scale: scale

Name	Value
General	
Unit	meters
Scale	10.0
Type	Defined graphically

A Appendix

KanbanExpeditePolicies

Name	Value
Length, pixels	100.0
Show at runtime	false
Lock	false
Public	false
Position and size	
x	0.0
y	-150.0
Rotation	0.0

Parameter: policyType

Name	Value
General	
Array	false
Default value	2
Type	int
Show at runtime	true
Show name	true
Value editor	
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: urgentThresholdHours

Name	Value
General	
Array	false
Default value	2281
Type	double
Show at runtime	true
Show name	true
Value editor	
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: durationMode

Name	Value
General	
Array	false
Default value	1
Type	int
Show at runtime	true
Show name	true
Value editor	

A Appendix

KanbanExpeditePolicies

Name	Value
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: preemptionOverheadHours

Name	Value
General	
Array	false
Default value	24
Type	double
Show at runtime	true
Show name	true
Value editor	
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: reservedUrgentServers

Name	Value
General	
Array	false
Default value	24
Type	int
Show at runtime	true
Show name	true
Value editor	
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: expediteWipLimit

Name	Value
General	
Array	false
Default value	24
Type	int
Show at runtime	true
Show name	true
Value editor	
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

A Appendix

KanbanExpeditePolicies

Parameter: useToyData

Name	Value
General	
Array	false
Default value	false
Type	boolean
Show at runtime	true
Show name	true
Value editor	
Editor control	Check Button
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: useTrim995

Name	Value
General	
Array	false
Default value	false
Type	boolean
Show at runtime	true
Show name	true
Value editor	
Editor control	Check Button
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: resultsFileName

Name	Value
General	
Array	false
Default value	"simulation_results.csv"
Type	String
Show at runtime	true
Show name	true
Value editor	
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: inputSeed

Name	Value
General	

A Appendix

KanbanExpeditePolicies

Name	Value
Array	false
Default value	0
Type	int
Show at runtime	true
Show name	true
Value editor	
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

Parameter: inputRunId

Name	Value
General	
Array	false
Default value	0
Type	int
Show at runtime	true
Show name	true
Value editor	
Editor control	Text
Advanced	
System dynamics units	false
Save in snapshot	true

Function: sampleServiceTime

Name	Value
General	
Return type	double
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>if(isUrgent) { int idx = uniform_discr(0, urgentServiceSamples.size() - 1); return urgentServiceSamples.get(idx); } else { int idx = uniform_discr(0, normalServiceSamples.size() - 1); return normalServiceSamples.get(idx); }</pre>
Advanced	
Access type	default
System dynamics units	false

Arguments:

Name	Type
isUrgent	boolean

A Appendix

KanbanExpeditePolicies

Function: findIdleServer

Name	Value
General	
Return type	Server
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>for(Server s : servers) { if(s.busy) { continue; } if(policyType == 3) { // Reserved Capacity if(s.reservedForUrgent) { if(urgentQueue.size() > 0) { return s; } } else { if(urgentQueue.size() > 0 normalQueue.size() > 0) { return s; } } } else { return s; } } return null;</pre>
Advanced	
Access type	default
System dynamics units	false

Function: startService

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre>s.busy = true; s.currentItem = item; item.serviceStartTime = time(); s.serviceCompletion.restart(item.remainingServiceTime);</pre>
Advanced	
Access type	default
System dynamics units	false

Arguments:

Name	Type
s	Server
item	WorkItem

Function: dispatchWork

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre>while(true) { Server s = findIdleServer(); if(s == null) { if(policyType == 2 && urgentQueue.size() > 0) { Server preemptable = findPreemptableServer(); if(preemptable != null) { preemptCurrentWork(preemptable); s = findIdleServer(); } } } if(s == null) { return; } WorkItem nextItem = selectNextItem(s); if(nextItem == null) { return; } startService(s, nextItem); }</pre>
Advanced	
Access type	default
System dynamics units	false

Function: selectNextItem

Name	Value
General	
Return type	WorkItem
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>if(policyType == 3) { // Reserved Capacity if(s.reservedForUrgent) { if(urgentQueue.size() > 0) { return urgentQueue.removeFirst(); } return null; } if(urgentQueue.size() > 0) { return urgentQueue.removeFirst(); } if(normalQueue.size() > 0) { return normalQueue.removeFirst(); } return null; }</pre>

A Appendix

KanbanExpeditePolicies

Name	Value
	<pre>} else if(policyType == 4) { // Strict Expedite WIP Limit if(urgentQueue.size() > 0 && countUrgentInService() < expediteWipLimit) { return urgentQueue.removeFirst(); } if(normalQueue.size() > 0) { return normalQueue.removeFirst(); } return null; } else if(policyType == 1 policyType == 2) { if(urgentQueue.size() > 0) { return urgentQueue.removeFirst(); } if(normalQueue.size() > 0) { return normalQueue.removeFirst(); } return null; } return null;</pre>
Advanced	
Access type	default
System dynamics units	false

Arguments:

Name	Type
s	Server

Function: findPreemptableServer

Name	Value
General	
Return type	Server
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>for(Server s : servers) { if(s.busy && s.currentItem != null && !s.currentItem.isUrgent) { return s; } } return null;</pre>
Advanced	
Access type	default
System dynamics units	false

Function: preemptCurrentWork

Name	Value
------	-------

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre> WorkItem interrupted = s.currentItem; if(interrupted != null) { return; } double workedTime = time() - interrupted.serviceStartTime; interrupted.remainingServiceTime = Math.max(0, interrupted.remainingServiceTime - workedTime) + preemptionOverheadHours; interrupted.numPreemptions++; totalPreemptions++; if(startupTimeCheck < 0) { startupTimeCheck = time(); } s.serviceCompletion.reset(); s.currentItem = null; s.busy = false; </pre>
Advanced	
Access type	default
System dynamics units	false

Arguments:

Name	Type
s	Server

Function: countUrgentInService

Name	Value
General	
Return type	int
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre> int count = 0; for(Server s : servers) { if(s.busy && s.currentItem != null && s.currentItem.isUrgent) { count++; } } return count; </pre>
Advanced	
Access type	default
System dynamics units	false

Function: injectOneItem

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre>if(isUrgent) { sourceUrgent.inject(1); } else { sourceNormal.inject(1); }</pre>
Advanced	
Access type	default
System dynamics units	false

Arguments:

Name	Type
isUrgent	boolean

Function: injectToyData

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre>injectArrivalBatch(10, 3);</pre>
Advanced	
Access type	default
System dynamics units	false

Function: injectArrivalBatch

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre>for(int i = 0; i < normalCount; i++) { injectOneItem(false); } for(int i = 0; i < urgentCount; i++) { injectOneItem(true); }</pre>
Advanced	
Access type	default
System dynamics units	false

Arguments:

A Appendix

KanbanExpeditePolicies

Arguments:

Name	Type
normalCount	int
urgentCount	int

Function: allServersIdle

Name	Value
General	
Return type	boolean
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>for(Server s : servers) { if(s.busy) { return false; } } return true;</pre>
Advanced	
Access type	default
System dynamics units	false

Function: allQueuesEmpty

Name	Value
General	
Return type	boolean
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>return normalQueue.size() == 0 && urgentQueue.size() == 0;</pre>
Advanced	
Access type	default
System dynamics units	false

Function: simulationCanFinish

Name	Value
General	
Return type	boolean
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>return allArrivalsLoaded && allServersIdle() && allQueuesEmpty();</pre>
Advanced	
Access type	default

A Appendix

KanbanExpeditePolicies

Name	Value
System dynamics units	false

Function: loadServiceSamples

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre>normalServiceSamples.clear(); urgentServiceSamples.clear(); while((useTrim995 ? serviceFileTrim995 : serviceFile).canReadMore()) { String line = (useTrim995 ? serviceFileTrim995 : serviceFile).readLine(); if(line == null) { continue; } String cleaned = line.replace("\\\"", "").trim(); if(cleaned.isEmpty()) { continue; } if(cleaned.startsWith("issue_key,")) { continue; } String[] parts = cleaned.split(",", 2); String itemClass = parts[1].trim().toLowerCase(); double serviceHours = Double.parseDouble(parts[2].trim()); if(itemClass.equals("urgent")) { urgentServiceSamples.add(serviceHours); } else if(itemClass.equals("normal")) { normalServiceSamples.add(serviceHours); } } println("Loaded service samples - normal: " + normalServiceSamples.size() + ", urgent: " + urgentServiceSamples.size())</pre>
Advanced	
Access type	default
System dynamics units	false

Function: calculatePercentile

Name	Value
General	
Return type	double
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>if(values == null values.isEmpty()) {</pre>

A Appendix

KanbanExpeditePolicies

Name	Value
	<pre>return Double.NaN; } ArrayList<Double> sorted = new ArrayList<Double>{ values }; Collections.sort(sorted); int n = sorted.size(); if(n == 1) { return sorted.get(0); } double rank = percentile * (n - 1); int lower = (int) Math.floor(rank); int upper = (int) Math.ceil(rank); if(lower == upper) { return sorted.get(lower); } double weight = rank - lower; return sorted.get(lower) + weight * (sorted.get(upper) - sorted.get(lower));</pre>
Advanced	
Access type	default
System dynamics units	false

Arguments:

Name	Type
values	ArrayList<Double>
percentile	double

Function: calculateUrgentSlaHitRate

Name	Value
General	
Return type	double
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>if(urgentLeadTimes == null urgentLeadTimes.isEmpty()) { return Double.NaN; } int hits = 0; for(double lt : urgentLeadTimes) { if(lt <= urgentThresholdHours) { hits++; } } return (double) hits / urgentLeadTimes.size();</pre>
Advanced	
Access type	default
System dynamics units	false

Function: getDurationVariantLabel

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Return type	String
Return type:	Returns value
Show at runtime	true
Show name	true
Function body	
Body	<pre>if(useTrim995) { return "work_hours_trim995"; } return "work_hours";</pre>
Advanced	
Access type	default
System dynamics units	false

Function: writeResultsHeaderIfNeeded

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre>java.io.File f = new java.io.File(resultsFileName); if(f.exists() && f.length() > 0) { return; } try { java.io.PrintWriter out = new java.io.PrintWriter(new java.io.FileWriter(f, false)); out.println("run_id," + "policyType," + "policyLabel," + "randomnessMode," + "durationVariant," + "useTrim995," + "numServers," + "urgentThresholdHours," + "preemptionOverheadHours," + "reservedUrgentServers," + "expediteWipLimit," + "normal_p50," + "normal_p85," + "normal_p95," + "normal_p95_minus_p50," + "urgent_sla_hit_rate," + "urgent_p50," + "urgent_p85," + "urgent_p95," + "totalCompleted," + "totalCompletedUrgent," + "totalUrgentMetThreshold," + "totalPreemptions," + "simulationFinishTime," + "normalLeadTimeCount," + "urgentLeadTimeCount," + "serviceSampleCountNormal," + "serviceSampleCountUrgent"</pre>

A Appendix

KanbanExpeditePolicies

Name	Value
	<pre>); out.close(); } catch(Exception e) { println("Failed to write results header: " + e.getMessage()); }</pre>
Advanced	
Access type	default
System dynamics units	false

Function: appendRunResult

Name	Value
General	
Return type:	Just action (returns nothing)
Show at runtime	true
Show name	true
Function body	
Body	<pre>double normalP50 = calculatePercentile(normalLeadTimes, 0.50); double normalP85 = calculatePercentile(normalLeadTimes, 0.85); double normalP95 = calculatePercentile(normalLeadTimes, 0.95); double normalSpread = normalP95 - normalP50; double urgentSlaHitRate = calculateUrgentSlaHitRate(); double urgentP50 = calculatePercentile(urgentLeadTimes, 0.50); double urgentP85 = calculatePercentile(urgentLeadTimes, 0.85); double urgentP95 = calculatePercentile(urgentLeadTimes, 0.95); try { java.io.PrintWriter out = new java.io.PrintWriter(new java.io.FileWriter(resultsFileName, true)); out.println(runId + "," + policyType + "," + "\"" + activePolicyLabel + "\",\" + "\"" + unique_random_seed + "\",\" + "\"" + getDurationVariantLabel() + "\",\" + useTrim995 + \",\" + numServers + \",\" + urgentThresholdHours + \",\" + preemptionOverheadHours + \",\" + reservedUrgentServers + \",\" + expediteWipLimit + \",\" + normalP50 + \",\" + normalP85 + \",\" + normalP95 + \",\" + normalSpread + \",\" + urgentSlaHitRate + \",\" + urgentP50 + \",\" + urgentP85 + \",\" + urgentP95 + \",\" + totalCompleted + \",\" + totalCompletedUrgent + \",\" + totalUrgentMetThreshold + \",\" + totalPreemptions + \",\" + time() + \",\" + normalLeadTimes.size() + \",\" + urgentLeadTimes.size() + \",\" + normalServiceSamples.size() + \",\" + urgentServiceSamples.size());</pre>

A Appendix

KanbanExpeditePolicies

Name	Value
	<pre>out.close(); } catch(Exception e) { println("Failed to append run result: " + e.getMessage()); }</pre>
Advanced	
Access type	default
System dynamics units	false

Event: weeklyArrivalEvent

Name	Value
General	
Logging	true
EVENT_TIMEOUT_PROPERTIES	- Recurring Event Properties
Mode	Cyclic
Trigger type	Timeout
Show at runtime	true
Show name	true
Action	
Action	<pre>if(!arrivalFile.canReadMore()) { allArrivalsLoaded = true; weeklyArrivalEvent.reset(); return; } String line = arrivalFile.readLine(); String cleaned = line.replace("\\", "").trim(); if(cleaned.startsWith("week,")) { return; } String[] parts = cleaned.split(","); int normalCount = Integer.parseInt(parts[1].trim()); int urgentCount = Integer.parseInt(parts[2].trim()); injectArrivalBatch(normalCount, urgentCount); println("Injected week: " + cleaned);</pre>

Event: finishCheckEvent

Name	Value
General	
Logging	true
Condition	simulationCanFinish()
Trigger type	Condition
Show at runtime	true
Show name	true
Action	
Action	<pre>println("Simulation can finish at time = " + time()); finishSimulation();</pre>

Variable: numServers

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Initial value	800
Type	int
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: totalCompleted

Name	Value
General	
Initial value	0
Type	int
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: totalCompletedUrgent

Name	Value
General	
Initial value	0
Type	int
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: totalUrgentMetThreshold

Name	Value
General	
Initial value	0
Type	int
Show at runtime	true
Show name	true
Advanced	
Access type	public

A Appendix

KanbanExpeditePolicies

Name	Value
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: lastCompletionTime

Name	Value
General	
Initial value	0
Type	double
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: totalPreemptions

Name	Value
General	
Initial value	0
Type	int
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: startupTimeCheck

Name	Value
General	
Initial value	-1
Type	double
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: activePolicyLabel

Name	Value
------	-------

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Initial value	"not set"
Type	String
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: allArrivalsLoaded

Name	Value
General	
Initial value	false
Type	boolean
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: normalServiceSamples

Name	Value
General	
Initial value	new ArrayList<Double>()
Type	ArrayList<Double>
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: urgentServiceSamples

Name	Value
General	
Initial value	new ArrayList<Double>()
Type	ArrayList<Double>
Show at runtime	true
Show name	true
Advanced	
Access type	public

A Appendix

KanbanExpeditePolicies

Name	Value
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: normalLeadTimes

Name	Value
General	
Initial value	new ArrayList<Double>()
Type	ArrayList<Double>
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: urgentLeadTimes

Name	Value
General	
Initial value	new ArrayList<Double>()
Type	ArrayList<Double>
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: runId

Name	Value
General	
Initial value	0
Type	int
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: currentSeed

Name	Value
------	-------

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Initial value	0
Type	int
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Queue: normalQueue

Name	Value
General	
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Show name	true
Advanced	
Show at runtime	true
Public	false
Logging	true

Agent Parameters:

Name	Value
Capacity	100000
Maximum capacity	false
Queuing	self.QUEUEING_FIFO
Enable exit on timeout	false
Enable preemption	false
Restore agent location on exit	true
Force statistics collection	false
On enter	dispatchWork();

Queue: urgentQueue

Name	Value
General	
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Show name	true
Advanced	
Show at runtime	true
Public	false

A Appendix

KanbanExpeditePolicies

Name	Value
Logging	true

Agent Parameters:

Name	Value
Capacity	100000
Maximum capacity	false
Queuing	self.QUEUING_FIFO
Enable exit on timeout	false
Enable preemption	false
Restore agent location on exit	true
Force statistics collection	false
On enter	dispatchWork();

Sink: doneSink

Name	Value
General	
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Show name	true
Advanced	
Show at runtime	true
Public	false
Logging	true

Agent Parameters:

Name	Value
On enter	if(simulationCanFinish()) { println("Simulation can finish at time = " + time()); appendRunResult(); finishSimulation(); }
Destroy policy:	com.anylogic.libraries.processmodeling.Sink.DESTROY_ON LY_CREATED_IN_SOURCE

Source: sourceNormal

Name	Value
General	
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Show name	true
Advanced	
Show at runtime	true
Public	false
Logging	true

A Appendix

KanbanExpeditePolicies

Agent Parameters:

Name	Value
Arrivals defined by	self.MANUAL
Location of arrival	self.LOCATION_NOT_SPECIFIED
New agent	new kanbanexpeditepolicies.WorkItem()
Change dimensions	false
Add agents to:	false
Forced pushing	true
On exit	agent.isUrgent = false; agent.arrivalTime = time(); agent.serviceTimeOriginal = sampleServiceTime(false); agent.remainingServiceTime = agent.serviceTimeOriginal; agent.urgentDueTime = agent.arrivalTime + urgentThresholdHours; agent.numPreemptions = 0; agent.metUrgentThreshold = false; agent.serviceStartTime = 0; agent.completionTime = 0;

Source: sourceUrgent

Name	Value
General	
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Show name	true
Advanced	
Show at runtime	true
Public	false
Logging	true

Agent Parameters:

Name	Value
Arrivals defined by	self.MANUAL
Location of arrival	self.LOCATION_NOT_SPECIFIED
New agent	new kanbanexpeditepolicies.WorkItem()
Change dimensions	false
Add agents to:	false
Forced pushing	true
On exit	agent.isUrgent = true; agent.arrivalTime = time(); agent.serviceTimeOriginal = sampleServiceTime(true); agent.remainingServiceTime = agent.serviceTimeOriginal; agent.urgentDueTime = agent.arrivalTime + urgentThresholdHours; agent.numPreemptions = 0; agent.metUrgentThreshold = false; agent.serviceStartTime = 0; agent.completionTime = 0;

Server: servers

Name	Value
------	-------

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Replication	numServers
Initialization Type	Contains a given number of agents
Population of agents	true
Replication	numServers
Initialization Type	Contains a given number of agents
Population of agents	true
Show name	true
Initial position	
Place agent(s)	in the specified point
Statistics	
Statistics	[]
Advanced	
Show at runtime	true
Public	false
Embedded object collection type	Access by index (ArrayList)
Logging	true

Enter: doneEnter

Name	Value
General	
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute:]
Show name	true
Advanced	
Show at runtime	true
Public	false
Logging	true

Agent Parameters:

Name	Value
New location	self.LOCATION_NOT_SPECIFIED
Change dimensions	false
Add newborns to:	false
Forced pushing	true

Enter: preemptedEnter

Name	Value
General	
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute: WorkItem]
Population of agents	false
Generic Parameters Substitutes	[Generic Parameter Substitute: WorkItem]
Show name	true
Advanced	

A Appendix

KanbanExpeditePolicies

Name	Value
Show at runtime	true
Public	false
Logging	true

Agent Parameters:

Name	Value
New location	self.LOCATION_NOT_SPECIFIED
Change dimensions	false
Add newborns to:	false
Forced pushing	true

Text File: arrivalFile

Name	Value
General	
Text file separators	[C@56fe2233
File mode	Read
Text file name	ResourceReference: kanbanexpeditepolicies./Users/[REDACTED]/Downloads/ThePublicJiraD ataset_SPARK/repro_out/2015- 1018/SPARK_arrivals_weekly_2015_2018_wide.csv (Resolved: true)
Text File Type	File
Show at runtime	true
Show name	true

Text File: serviceFile

Name	Value
General	
Text file separators	[C@28790ede
File mode	Read
Text file name	ResourceReference: kanbanexpeditepolicies./Users/[REDACTED]/Downloads/ThePublicJiraD ataset_SPARK/repro_out/2015- 1018/SPARK_service_workhours_2015_2018_clean.csv (Resolved: true)
Text File Type	File
Show at runtime	true
Show name	true

Text File: serviceFileTrim995

Name	Value
General	
Text file separators	[C@31d07dd9
File mode	Read
Text file name	ResourceReference: kanbanexpeditepolicies./Users/[REDACTED]/Downloads/ThePublicJiraD ataset_SPARK/repro_out/2015- 1018/SPARK_service_workhours_2015_2018_clean_trim995.csv
Text File Type	File

A Appendix

KanbanExpeditePolicies

Name	Value
Show at runtime	true
Show name	true

Link to agents: connections

Name	Value
General	
Show at runtime	true
Show name	true
Communication	
Message type	Object
Animation	
Draw line	false

Agent Type: WorkItem

Name	Value
Agent in flowcharts	
Use in flowcharts as	Agent
Dimensions and movement	
Speed	(10 : MPS)
Rotate animation towards movement	true
Rotate vertically (incline on ramps)	false
Space and network	
Space Type	Continuous
Layout Type	User-defined
Network type	User-defined
Enable steps	false
Advanced Java	
Generic	false
Omniverse	
Usd binding type	None
Advanced	
Flowchart block	false
Logging	true
Auto-create datasets	true
AOC_DATASETS_UPDATE_TIME_PROPERTIES	- Recurring Event Properties
Limit the number of data samples	false

A Appendix

KanbanExpeditePolicies

 connections

-  isUrgent
-  arrivalTime
-  serviceTimeOriginal
-  remainingServiceTime
-  completionTime
-  urgentDueTime
-  numPreemptions
-  serviceStartTime
-  metUrgentThreshold

Scale: scale

Name	Value
General	
Unit	meters
Scale	10.0
Type	Defined graphically
Length, pixels	100.0
Show at runtime	false
Lock	false
Public	false
Position and size	
x	0.0
y	-150.0
Rotation	0.0

Variable: isUrgent

Name	Value
General	
Initial value	false
Type	boolean
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: arrivalTime

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Initial value	0
Type	double
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: serviceTimeOriginal

Name	Value
General	
Initial value	0
Type	double
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: remainingServiceTime

Name	Value
General	
Initial value	0
Type	double
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: completionTime

Name	Value
General	
Initial value	0
Type	double
Show at runtime	true
Show name	true
Advanced	
Access type	public

A Appendix

KanbanExpeditePolicies

Name	Value
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: urgentDueTime

Name	Value
General	
Initial value	0
Type	double
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: numPreemptions

Name	Value
General	
Initial value	0
Type	int
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: serviceStartTime

Name	Value
General	
Initial value	0
Type	double
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: metUrgentThreshold

Name	Value
------	-------

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Initial value	false
Type	boolean
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Link to agents: connections

Name	Value
General	
Show at runtime	true
Show name	true
Communication	
Message type	Object
Animation	
Draw line	false

Agent Type: Server

Name	Value
Agent in flowcharts	
Use in flowcharts as	Agent
Dimensions and movement	
Speed	(10 : MPS)
Rotate animation towards movement	true
Rotate vertically (incline on ramps)	false
Space and network	
Space Type	Continuous
Layout Type	User-defined
Network type	User-defined
Enable steps	false
Advanced Java	
Generic	false
Omniverse	
Usd binding type	None
Advanced	
Flowchart block	false
Logging	true
Auto-create datasets	true
AOC_DATASETS_UPDATE_TIME_PROPERTIES	- Recurring Event Properties
Limit the number of data samples	false

A Appendix

KanbanExpeditePolicies

 main

 connections

 busy

 serviceCompletion

 currentItem

 reservedForUrgent

Scale: scale

Name	Value
General	
Unit	meters
Scale	10.0
Type	Defined graphically
Length, pixels	100.0
Show at runtime	false
Lock	false
Public	false
Position and size	
x	0.0
y	-150.0
Rotation	0.0

Event: serviceCompletion

Name	Value
General	
Logging	true
Timeout	(1 : HOUR)
Mode	User control
Trigger type	Timeout
Show at runtime	true
Show name	true
Action	
Action	<pre>WorkItem finished = currentItem; finished.completionTime = time(); main.lastCompletionTime = finished.completionTime; double leadTime = finished.completionTime - finished.arrivalTime; if(finished.isUrgent) { main.urgentLeadTimes.add(leadTime); } else { main.normalLeadTimes.add(leadTime); } main.totalCompleted++;</pre>

A Appendix

KanbanExpeditePolicies

Name	Value
	<pre>if(finished.isUrgent) { finished.metUrgentThreshold = finished.completionTime <= finished.urgentDueTime; } if(finished.isUrgent) { main.totalCompletedUrgent++; if(finished.metUrgentThreshold) { main.totalUrgentMetThreshold++; } } main.doneEnter.take(finished); currentItem = null; busy = false; main.dispatchWork();</pre>

Variable: busy

Name	Value
General	
Initial value	false
Type	boolean
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: currentItem

Name	Value
General	
Initial value	null
Type	WorkItem
Show at runtime	true
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Variable: reservedForUrgent

Name	Value
General	
Initial value	false
Type	boolean
Show at runtime	true

A Appendix

KanbanExpeditePolicies

Name	Value
Show name	true
Advanced	
Access type	public
Constant	false
Save in snapshot	true
System dynamics units	false

Link to agents: connections

Name	Value
General	
Show at runtime	true
Show name	true
Communication	
Message type	Object
Animation	
Draw line	false

Simulation Experiment: Simulation

Name	Value
General	
Bypass Initial Simulation Screen	true
Maximum available memory	512
Agent type	Main
Model time	
Execution mode	Real time with scale
Real time scale	1.0
Stop option	Never
Initial time	0.0
Initial date	Fri Apr 10 00:00:00 GMT 2026
Randomness	
Random Number Generation Type	Fixed seed (reproducible simulation runs)
Seed value	2
Window	
Title	KanbanExpeditePolicies : Simulation
Enable zoom and panning	true
Enable developer panel	true
Show developer panel on start	false
Advanced	
Load root from snapshot	false

KanbanExpeditePolicies

Text: text

Name	Value
------	-------

A Appendix

KanbanExpeditePolicies

Name	Value
General	
Show at runtime	true
Lock	false
Text	
Text	KanbanExpeditePolicies
Appearance	
Color	royalBlue
Alignment	LEFT
Position and size	
x	50.0
y	30.0
z	0.0
Rotation	0.0
Omniverse	
Usd binding type	None
Advanced	
Show name	false

Parameter Variation Experiment: PolicyBatch

Name	Value
General	
Maximum available memory	512
Agent type	Main
Model time	
Stop option	Never
Initial time	0.0
Initial date	Fri Apr 10 00:00:00 GMT 2026
Randomness	
Random Number Generation Type	Random seed (unique simulation runs)
Replications	
Use replications	true
Fixed replications number	true
Replications per iteration	10000
Window	
Title	KanbanExpeditePolicies : PolicyBatch
Enable zoom and panning	true
Enable developer panel	true
Show developer panel on start	false
Advanced	
Allow parallel evaluations	false
Load root from snapshot	false

Parameter Variation Experiment Parameters:

Parameter	Value
policyType	index + 1
urgentThresholdHours	
durationMode	

A Appendix

KanbanExpeditePolicies

Parameter Variation Experiment Parameters:

Parameter	Value
preemptionOverheadHours	
reservedUrgentServers	
expediteWipLimit	
useToyData	
useTrim995	
resultsFileName	"policy_batch_800.csv"
inputSeed	1
inputRunId	index + 1