UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering

Jannis Rosenbaum

Optimization of Activity Batching Policies in Business Processes

Master's Thesis (30 ECTS)

Supervisors:
Orlenys López-Pintado, PhD
Marlon Dumas, PhD

Tartu 2025

Optimization of Activity Batching Policies in Business Processes

Abstract:

Many business processes group similar tasks into batches to reduce costs and improve efficiency. Batching allows managers to trade off cost and processing effort against waiting time. Larger and less frequent batches can reduce effort by amortizing fixed costs, but they increase waiting times. Conversely, smaller and more frequent batches minimize waiting times but lead to higher processing effort and increased fixed costs. A batching policy defines how tasks are grouped and when each group is executed. This thesis addresses the problem of discovering batching policies that achieve the best balance between waiting time, processing effort, and cost. It proposes a Pareto optimization approach that begins with a given set (possibly empty) of batching policies and generates alternatives using intervention heuristics. Each heuristic identifies an opportunity to improve a policy with respect to a specific metric (e.g., waiting time or resource utilization) and suggests an adjustment. The effects of these interventions are evaluated via simulation. The intervention heuristics are embedded into optimization meta-heuristics — hill climbing, simulated annealing, and reinforcement learning — that iteratively refine the Pareto front of discovered solutions. An experimental evaluation compares the heuristic-driven approach to non-guided baselines, assessing convergence, diversity, and cycle time improvements. The result is a set of optimized batching strategies that present different trade-offs between speed, cost, and resource usage — empowering decision-makers to select the policy that best fits their operational needs. Additionally, a graphical user interface was developed to facilitate intuitive access to the optimizer software and improve the readability of the results.

Keywords: Business Processes, Business Process Optimization, Activity Batching Policy, Batching Policy Optimization

CERCS: P170

Tegevuste rühmitamispoliitikate optimeerimine äriprotsessides

Lühikokkuvõte:

Paljud äriprotsessid rühmitavad sarnaseid ülesandeid rühmitamise teel, et vähendada kulusid ja tõsta tõhusust. Rühmitamine võimaldab juhtidel tasakaalustada kulusid ja töötlemiskulusid ooteajaga. Suuremad ja harvemad rühmitamised aitavad vähendada kulusid, amortiseerides fikseeritud kulusid, kuid pikendavad ooteaega. Vastupidi, väiksemad ja sagedasemad rühmitamised minimeerivad ooteaega, kuid suurendavad töötlemiskulusid ja fikseeritud kulusid. Rühmitamispoliitika määrab, kuidas ülesandeid grupeeritakse ja millal iga grupp viiakse läbi.

See lõputöö käsitleb rühmitamispoliitika avastamise probleemi, mis saavutavad parima tasakaalu ooteaja, töötlemiskulude ja kulude vahel. Töö pakub Pareto-optimeerimise lähenemist, mis algab antud (või tühja) rühmitamispoliitikate kogumiga ja genereerib alternatiive sekkumisheuristikate abil. Iga heuristik tuvastab võimaluse poliitika parandamiseks konkreetse mõõdiku (nt ooteaeg või ressursside kasutus) osas ning soovitab vastava kohanduse. Nende sekkumiste mõju hinnatakse simulatsiooni kaudu.

Sekkumisheuristikad on integreeritud optimeerimise metaheuristikatesse — hill climbing, simulated annealing ja reinforcement learning —, mis iteratiivselt täiustavad leitud lahenduste Pareto-fronti. Eksperimentaalne võrdlus hindab heuristikapõhise lähenemise ja juhendamata baasjoonte konvergentsi, mitmekesisust ja tsükliaja paranemist.

Tulemuseks on optimeeritud rühmitamisstrateegiate kogum, mis esitab erinevaid kompromisse kiiruse, kulude ja ressursside kasutamise vahel — võimaldades otsustajatel valida just oma operatsiooniliste vajadustega parima poliitika. Lisaks töötati välja graafiline kasutajaliides, mis võimaldab intuitiivset ligipääsu optimeerimistarkvarale ja parandab tulemuste loetavust.

Võtmesõnad: Äriprotsessid, äriprotsesside optimeerimine, tegevuste pakkimispoliitika, pakkimispoliitika optimeerimine

CERCS: P170

Contents

1. Introduction 6
2. Background 8
    2.1 Business Process 8
    2.2 Business Process Modeling Notation 8
        2.2.1 Basic Symbols 9
        2.2.2 Example Process 10
        2.2.3 Key Performance Indicators 11
        2.2.4 Batching 12
    2.3 Business Process Simulation 13
    2.4 Business Process Optimization 15
        2.4.1 Pareto Fronts 15
        2.4.2 Hill Climbing 16
        2.4.3 Simulated Annealing 16
        2.4.4 Proximal Policy Optimization 16
    2.5 Architecture Patterns 17
3. Related Work 19
4. Approach 21
    4.1 Batching Model 21
    4.2 Intervention Heuristics 23
    4.3 Optimization Description 29
        4.3.1 Hill-Climbing and Simulated Annealing Optimization 31
        4.3.2 Reinforcement Learning for Batching Policies 33
    4.4 Software Architecture 34
5. Evaluation & Testing 38
    5.1 Datasets and Experimental Setup 38
    5.2 Metrics 39
    5.3 Results 40
    5.4 Static Analysis & Testing 43
6. Graphical User Interface 46
    6.1 Architecture 46
    6.2 Usage 47
7. Conclusion 50

References 51
Appendices 56
License 59

1. Introduction

Every organization needs to perform certain activities, react to events, and make decisions to deliver value to a customer. A *process* or more precisely a *Business Process* (*BP*) is the structured sequence of these activities, events, and decisions. *BPs* also involve *resources* such as machines, information systems, or humans, which perform activities, react to events, and make decisions [1].

A common practice in many *BPs* is *batching*. It involves grouping multiple instances of the same activity for execution [2]. *Batching* can improve efficiency, optimize resource utilization, and reduce operational costs and processing times, but it can also be necessary due to logistical constraints or operational requirements. Nevertheless, batching introduces a trade-off: larger batches reduce per-unit costs by amortizing fixed fees and decreasing processing times through higher concurrency. However, they also increase waiting times and delay the execution of individual tasks. In contrast, smaller batches reduce waiting time, since tasks are executed sooner, but may also raise costs due to more frequent setups and increase processing times by reducing concurrency and underutilizing resources.

For example, we can see this trade-off in hospital laboratories, where blood samples are tested in batches. Testing each sample as soon as it arrives gives fast results but is costly because the machines operating expenses are not shared across multiple samples. On the other hand, waiting to process a full batch lowers the cost per test but makes patients wait longer, which can delay medical decisions. Similar trade-offs occur in other industries: factories batch production to reduce machinery setup times and costs, meaning individual orders wait longer before they are processed. Logistics companies combine shipments to save on transport costs but risk delivery delays, and banks process loan applications in batches to speed up verification but extend approval times.

Existing approaches to batching optimization mainly focus on determining the optimal batch size, often overlooking a broader policy to activate the batches. A *batching policy* is not limited to batch size alone. It may also require multiple activation rules, such as time-based triggers (e.g., executing a batch every Monday at 10:00 AM) or constraints on how long a batch should wait before execution, from the arrival of the first or last activity instance. These activation rules may better align batching with workload peaks and resource schedules, ensuring batches form efficiently during high-demand periods and are processed when resources are available.

For practical reasons, *batching policies* should be readable and adaptable. A white-box policy allows analysts to interpret and adjust batching rules, helping organizations refine policies as workload patterns, resources, and business requirements evolve. Previous research in process mining has focused on discovering white-box batch processing patterns and identifying inefficiencies. However, a gap remains in data-driven, automated optimization of white-box *batching policies* to improve process performance. A key challenge is balancing conflicting objectives—reducing overall costs and processing times without increasing waiting times. Since exploring all possible policies is infeasible due to the complexity of the search space, we need heuristic methods to identify the factors impacting performance and apply targeted interventions that improve *batching policies*.

To address the gap, this paper presents an automated data-driven multi-objective optimization approach to minimize cycle time and cost by dynamically adjusting *batching policies*. Formulated as a *Pareto* optimization problem, it discovers non-dominated solutions by analyzing event logs to identify inefficiencies. It develops a set of 19 heuristic interventions to identify problematic scenarios and improvement actions, grouped into four categories: waiting time, processing time, cost efficiency, and resource utilization. Via *simulation*, it evaluates the impact of each intervention on cycle time and cost, iteratively updating the *Pareto* front. The approach integrates three meta-heuristics: Hill-Climbing explores local optima within a fixed radius, Simulated Annealing escapes local optima by probabilistically accepting worse solutions, and Reinforcement Learning learns optimal batching strategies from historical data. Experiments show that the approach improves cycle time and cost efficiency while outperforming a random baseline in convergence, spread, and distribution of *Pareto* fronts.

The implemented system combining these heuristics with the meta-heuristics is named *Optimos v2*, after the predecessor project *Optimos* [3], which focused on resource optimization and did not consider batching.

Experiments show that, for each of these meta-heuristics, the proposed heuristic interventions lead to better *Pareto fronts* than the same meta-heuristic coupled with a standard random perturbation function over neighbor solutions, with respect to well-accepted quality metrics for *Pareto fronts* (convergence, diversity, cycle-time gain per process case).

Additionally, in line with the thesis' white-box approach, a *Graphical User Interface* (*GUI*) was developed to enable even non-technical users to optimize their *BPs* batching policies in a user-friendly and human-readable way.

2. Background

This chapter defines key concepts of *Business Processes*, the *Business Process Model and Notation*, and *Business Process Optimization*. Furthermore, it introduces essential patterns relevant for understanding the architecture of *Optimos v2*

2.1 Business Process

As already introduced in chapter 1, *Business Processes* (*BPs*) are structured sequences of activities, events and decisions made in the context of the activities of resources in an organization that aim to provide value to customers [1]. This definition can be interpreted very broadly, and includes, for example, government agencies that provide services to the public. An individual in the organization, who manages and oversees the process is called a *process owner* [1].

A *BP* is initially an abstract concept. To be shared or improved, it must be modeled – either via textual descriptions or using a formal modeling language such as *Business Process Model and Notation* (*BPMN*)[4]. This thesis specifically targets the optimization of *BPs* modeled in *BPMN*, due to its widespread adoption and intuitive graphical representation it can be understood by technical, as well as non-technical roles in a company.

2.2 Business Process Modeling Notation

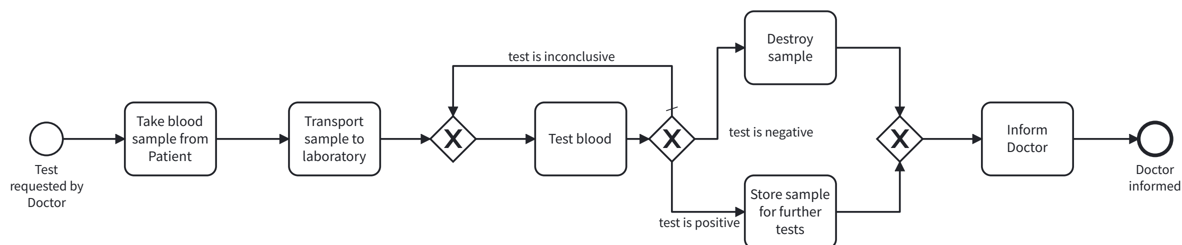
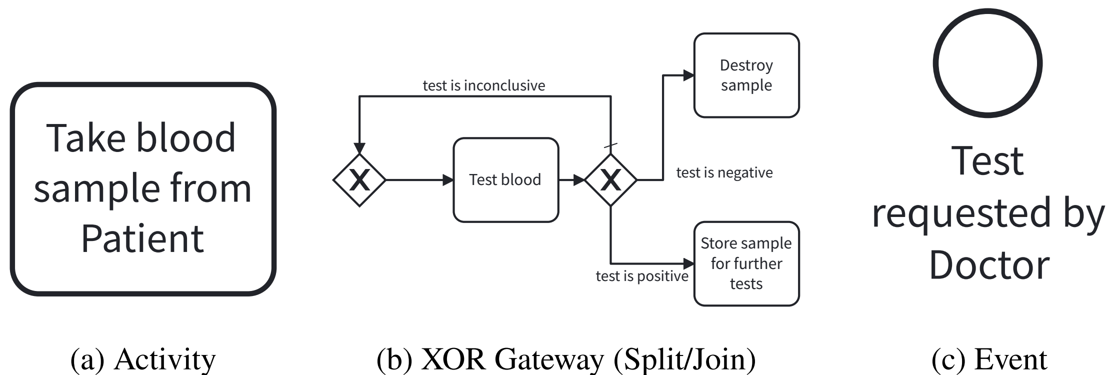


Figure 1. Example BPMN model of a blood testing process in a hospital setting.

The *Business Process Model and Notation* (*BPMN*) is a modeling language used to coherently represent *BPs* [4]. The language consists of well over 100 standardized elements and can be used to represent even highly complex *BPs* [1]. While BPMN models are typically visualized graphically (as shown in fig. 1), the *BPMN Specification*[4] also specifies a corresponding *Extensible Markup Language* (*XML*) based format. This *XML* encodes the process as a directed graph. As an exhaustive description of the *BPMN* is not the focus of this thesis, we focus on the most relevant elements illustrated via the running example

A representation (model) of the example process (section 2.2.2) in *BPMN* can be found in fig. 1.

2.2.1 Basic Symbols



(a) Activity (b) XOR Gateway (Split/Join) (c) Event

Figure 2. Basic BPMN symbols: (a) Activity, (b) XOR Gateway (Split/Join), and (c) Event.

The example graph in fig. 1 already introduces several fundamental *BPMN* elements, which are described by the *Business Process Model and Notation (BPMN) Specification, Version 2.0.2* [4] as follows:

I. **Activities:** An Activity represents work performed in the process. The smallest, atomic, type of Activity, that can not be decomposed more is a *Task*, typically performed by a resource. They are drawn with rounded corners, see fig. 2 a).

II. **Events:** An event denotes something that *happens* in the process. Events are either triggered by a cause or result in an impact. While events can also be used in the middle of a process, the most common are the start event, which triggers a new process instance, and the end event, which marks its completion. They are drawn as circles, e.g. refer to fig. 2 b).

III. **Arcs:** *Events*, *activities* and others are connected by directed arcs. The most common type is the *sequence flows*, which just define that one element follows another (i.e. the control or execution flow). *Sequence Flows* are represented as a line with a full arrow-head.

IV. **Gateways:** *Gateways* model the divergence or convergence of the process execution. The Gateway shown in fig. 2 b) is a *Exclusive OR* (*XOR*) gateway, that determines the next activity based on a condition. Similar gateways are also available for parallel execution, called *AND* gateways. Gateways are modeled as diamond shapes, and specifically, *XOR*-Gateways are depicted as diamond-shapes with an $\mathcal{X}$,while *AND*-gateways use a $+$ sign.

All of those elements, and all other *BPMN* elements have a graphical notation for visualization or rendering, but also an accompanying machine-readable *XML* schema [1]. This ensures that a *BPMN* modeled or generated within a tool, can be interchanged between different modeling tools (typically saved in a `.bpmn` text file). Although not every *XML* attribute has a graphical equivalent, most practical use cases allow a one-to-one mapping between the XML and graphical representations.

2.2.2 Example Process

Extending upon the brief example of the chapter 1, we examine the *BPs* of collecting and testing blood for a certain disease in a hospital setting. This running example will be used throughout the thesis to illustrate key concepts.

Blood is first taken from the patient by the doctor, then transported to the lab, where it is tested. Depending on the test result, it is either…

A) …retested if the result is inconclusive.

B) …stored for further analysis if the test was positive.

C) …disposed of if the test was negative.

If the result is inconclusive (case A), the test is repeated until a conclusive result is obtained. Once either case B or C is reached, the doctor is informed of the result.

Three resources are involved in this process: the *Doctor* (*Doc*), who collects the blood sample; the *Courier* (*Co*) who transports it from the patient wing to the laboratory; and the *Lab Technician* (*LT*) who performs the test.

The machine performing the tests might process up to ten samples at the same time, but requires a *LT* to supervise a running test at all times.

Business Processes are typically executed repeatedly, either sequentially or in parallel. In this example, each patient triggers a new instance of the same process—called a *case*—which proceeds independently through the activities of collection, transport, and testing.

[1] https://www.omg.org/spec/BPMN/20100501/BPMN20.xsd

2.2.3 Key Performance Indicators

To effectively analyze and improve *BPs* it is essential to identify the relevant *Key Performance Indicator* (*KPI*) for a process execution. Dumas et al. identify four different dimensions of *KPIs*: time, cost, quality and flexibility [1].

For the data-driven, quantitative analysis central to our approach, we are concentrating on the first two dimensions, as they can be expressed numerically, compared objectively, and collected via simulation (more on simulation in section 2.3). Those dimensions, time and cost, can be measured through various *KPIs*, but for our approach we focus on the following activity-level indicators.

1. Time
   - 1*a*) ***Processing Time* (*PT*)**: The *Processing Time* is the time it takes from the start of an activity to its end. For example, if the *LT* starts the blood test at 12:00 and finishes it at 12:30, the *PT* is 30 minutes.
   - 1*b*) ***Waiting Time* (*WT*)**: The *Waiting Time* is the time an activity waits to be started, after its predecessor (e.g. an activity or event) in the control flow has completed (known as *enablement* of the activity). For example, a blood test activity instance will must wait if all *LTs* are occupied doing other tests, or it will need to wait if not enough other samples are available to fill the machine fully.
   - 1*c*) ***Idle Time* (*IT*)**: The *Idle Time* is the time an activity is processing was paused due to resource unavailability. For example, if a blood test was started right before lunch break, the test will idle until an *LT* resumes the test.
   - 1*d*) ***Cycle Time* (*CT*)**: The *Cycle Time* is the total amount of time it takes for one activity instance from its enablement until the activity is completed. For example, if a sample arrives at the laboratory at 14:00 (thereby *enabling* the "Test blood" activity), the samples test is started at 15:00, and it finishes at 16:00, the *CT* is 2 hours.
2. **Cost**: *Cost* refers to the monetary cost needed to run an activity instance. It usually contains a fixed part (e.g. for machinery, rent, etc.) and a resource-dependent component (e.g., labor cost).

These metrics are defined per activity instance but can be aggregated across process instances – for example, by averaging the waiting time per activity.

Additionally, we refer to *resource utilization* throughout this thesis. It is measured by dividing the time a resource is performing activities – i.e., when a resource is actively *"working"* – by its

total available time. The availability is defined by the resources *calendar*. For example, a *Doc* working from 9:00 to 17:00 from Monday to Friday is available $40\,\mathrm{h/week}$; if they take 15 blood samples per day, each requiring $20\,\mathrm{min}$, then the total working time is $15 \times 20\,\mathrm{min} \times 5\,\mathrm{d/week} = 25\,\mathrm{h/week}$. This yields a resource utilization for the *Doc* of $\frac{25\,\mathrm{h/week}}{40\,\mathrm{h/week}} = 0.625$. Utilization is typically expressed as a percentage (e.g., 62.5%), with higher values generally indicating more efficient resource use.

For details on how these *KPIs* are applied in our approach, see section 4.3.

2.2.4 Batching

Batching is a scheduling mechanism that groups and executes multiple instances of the same activity together in a business process [2]. This is to be distinguished from *multitasking*, where resources perform *different* activities at the same time.

Revisiting our example (section 2.2.2), batching could be introduced by running multiple tests simultaneously. What impact does this have on the metrics defined in section 2.2.3?

The *Processing Time* decreases, as the ten samples will be processed in parallel by the machine, thereby resulting in a *Processing Time* that is close to that of processing a single sample. However, it is not exactly *equal* to that *PT* of one sample, since operations like inserting and removing ten samples will still take more time than just for one sample. Thus, the total *PT* of the batch is the *PT* of a single sample plus a batch-size-dependent overhead.

A similar observation applies to the *cost* of a batch: it may remain largely the same if we compare running the machine with one sample or ten samples e.g., in electricity consumption or machine wear and tear – whereas some costs like individual Petri dishes scale with the number of samples. This thesis refers to the former as *fixed cost*, and the latter as part of the *cost function*. However, a holistic analysis of the cost also includes personnel costs e.g., by multiplying the hourly wage of a given resource with the time taken for the batch, which in turn depends on the fixed and variable *Processing Time* components discussed earlier.

In summary, introducing batching reduces the *PT* and Cost in this example, but neither is equivalent to a single-instance execution. These additional costs (in time and actual monetary cost) will be referred to more generally as *cost functions* and are more formally defined in definition 2 (section 4.1).

Naturally, batching is not without drawbacks, as an analysis of the *Waiting Time* reveals. For example, if three blood samples are collected per day, but the batch size is ten, the first samples

collected might wait several days until enough samples are accumulated for one test run, which may be operationally impractical. So an alternative approach might be to run the machine partially filled, or to apply more complex rules, such as triggering execution when the batch is full or when the first sample has waited more than two days.

The increase in WT is also why it is not possible to generally state the impact of batching on the CT. Whether CT increases or decreases depends on the balance between reduced PT and increased WT. The impact on IT is similarly context-dependent; it may go down, if batching causes fewer tests to idle during the non-working hours (i.e. they were processed before end of the day), but conversely more samples per test also means that if idling happens, it happens to more samples at the same time.

Thus, to capture realistic business processes, we must combine multiple different batching rules of different types [5]. This collection of rules is referred to as a *batching policy*. A list and formalization of batching policy rules can be found in definition 3 (section 4.1).

Even in the relatively simple example, the complexity of identifying an optimal batching policy becomes apparent. Especially if we take into account multiple batches in combination; for instance, if sample transport is also batched, how long should a sample wait before dispatch? Should the *Co* transport ten samples at once, or fewer to preserve machine capacity for potential retests? These are all decisions a *process owner* might need to make and which are very hard to just do on human ”intuition” or static, non-data-driven process analysis.

2.3 Business Process Simulation

Business Process Simulation (*BPS*) is a widely used technique for quantitatively analyzing business processes and their modifications. *BPS* will take a *Business Process Model* (*BPM*), along with simulation parameters as input and then simulate multiple process instances (*cases*) at the same time. With this method, the *process owner* can experiment with changes on this *digital twin* and evaluate various *what-if* scenarios. *BPS* is thereby at the core of this approach, as it allows iterative testing of changes to batching policies through re-simulation

The aforementioned simulation input including the *BPM* for the control flow (e.g. a *BPMN-XML*-file), and the *simulation parameters* is referred to as the *simulation model*. The simulation

parameters allow to align the simulated process more closely with real-world behavior. *Prosimos*[2], used by *Optimos v2*, is an open source *BPS* engine, that supports differentiated resources, meaning each resource is defined individually. For example, each resource can have its own task-specific performance, its individual cost and its own availability calendar (*working hours*), in contrast to traditional models that use generic resource *pools*. Also, the *inter-arrival* time of new process cases (i.e. how often blood tests are requested), branching probabilities (i.e. likelihood of a sample retest) and *case attributes* (e.g. how urgent is a blood test) can be specified as fixed values or drawn from probability distributions (e.g., normal, exponential, or uniform).

Central for this thesis is the *batching policy* simulation parameter, which defines when an activity should be batched and what batch size should be used. A formal definition is provided in section 4.1.

If the parameters are not known to the user, or were never defined for a given process, it is possible to derive a simulation model using *process mining*. The input for a process mining procedure is an *event log*, these event logs can be collected by different ways, e.g., e.g., directly from a *Business Process Management System* (*BPMS*) or indirectly via logs of other enterprise systems or other software applications. An Event in its simplest form typically consists of an *activity*, the *resource* which performed it, a *start time*, and an *end time*. This log can therefore be represented in simple *Comma-Separated Values* (*CSV*) files or, for more complex scenarios, in *eXtensible Event Stream* (*XES*) format [1, 6].

The simulation models supported by *Prosimos*, and thereby *Optimos v2*, can be mined from *CSV*-based event logs using the *simod*[3] tool.

The output of *BPS* includes a simulated *event log* and statistical data, such as activity durations, waiting times, resource utilization, and more. We refer to this collection of metrics as an *evaluation*. The role of *BPS* within *Optimos v2*'s architecture is discussed in section 4.3.

[2] https://github.com/AutomatedProcessImprovement/Prosimos
[3] https://github.com/AutomatedProcessImprovement/Simod

2.4 Business Process Optimization

A common application of *Business Process Simulation* is *Business Process Optimization* (*BPO*). Through a cycle of simulations and mutations performed on the simulation parameters or the *BPMN* graph, one can find better (*more optimal*) variants of the process [3].

Finding *the* globally optimal solution —– i.e., the best possible combination of mutations such as resource calendars, resource assignments, or batching rules, as considered in this thesis —– is computationally intractable in practice. This is because the number of possible permutations (the *search space*) grows exponentially, making exhaustive evaluation prohibitively expensive. In fact, many underlying problems, such as finding an optimal batching strategy with constraints, are proven to be *NP-hard* [7], meaning that no algorithm is known to solve them optimally within reasonable (*polynomial*) time for larger instances.

Therefore, *BPO* techniques often rely on meta-heuristic-methods to iteratively approximate near-optimal solutions. Common meta-heurstics include *Hill-Climbing* and *Simulated-Annealing*, which explore the search space using guided randomness rather than exhaustive enumeration. Further details are provided in sections 2.4.2 and 2.4.3

Another trend in the optimization space is the use of *deep learning* models to discover patterns or correlations between the simulation model and its resulting *KPIs* that might be difficult to model with traditional declarative algorithms [8]. This thesis employs one such algorithm: *Proximal Policy Optimization* (*PPO*), a *deep*, *Reinforcement Learning* (*RL*) method; described in more detail in section 2.4.4.

2.4.1 Pareto Fronts

As discussed in section 2.2.4, introducing batching to process, brings benefits in terms of *PT* and reduced *cost*, but may also introduce trade-offs, such as increased *WT*.

For our approach to treat both the original and modified solution as equally valid trade-offs, we employ the concept of *Pareto fronts*:

For any given two n-dimensional solution evaluations A and B, B is *Pareto dominated* by A if A is strictly better than B in at least one dimension and A is at least as good as B in the remaining dimensions. A set of non-dominated solution evaluations is called a *Pareto set*. The set which solutions are not dominated by any other solutions is called *Pareto optimal*. The set of the evaluations of this *Pareto optimal* set is the *Pareto front* [9].

For example, assume we have three batching policies A, B, C with corresponding evaluations in dimensions (time, cost) as $E_A = (5, 5)$, $E_B = (4, 6)$ and $E_C = (4, 4)$, then E_C is *Pareto dominated* by both E_A and E_B. While E_A is better than E_B in the first evaluation dimension (time), it is worse in the second dimension (cost), therefore neither E_A is *Pareto dominated* by E_B, nor E_B by E_A. This makes $\{A, B\}$ a *Pareto set* and more specifically, the *Pareto optimal set* and $\{E_A, E_B\}$ is the *Pareto Front*. In this case, *Optimos v2* presents batching policies A and B along with their evaluations, enabling the analyst to select the solution best suited for their business context.

This is also known as *multi-parameter* or *multi-objective* optimization, as the optimizer is optimizing two parameters (time and cost) at the same time, in contrast to traditional single-objective optimization, which focuses on a single dimension or *KPI*, such as *Cycle Time*.

2.4.2 Hill Climbing

Hill-Climbing (*HC*) is a local-search optimization technique that greedily selects the best possible solution within its neighborhood. Thereby it converges quickly, but the solution may represent only a local optimum [9]. Originally, *HC* only optimizes a single dimension [10], but, with the extension proposed by Weise, it is also applicable to the multi-objective optimization of a *Pareto front* [11].

A detailed explanation of how *HC* is applied in our approach is provided in section 4.3.1.

2.4.3 Simulated Annealing

Simulated-Annealing (*SA*) [12] is a global-search optimization technique that randomly selects a solution from all possible neighbors in a given radius – based one the *temperature*. This *temperature* decreases over time, gradually narrowing the search to more local neighborhoods. It also probabilistically accepts worse-than-current solutions, with higher temperatures increasing the acceptance likelihood. It converges more slowly than *HC*, but is less prone to end up in a local optimum [10].

A more detailed description of how our approach uses *SA* can be found in section 4.3.1.

2.4.4 Proximal Policy Optimization

RL is an adaptive search method in which an agent interacts with a simulation: it observes the current state, takes an action and receives a reward, gradually learning which actions lead to better overall performance – unlike *HC* and *SA*, which treat each step independently. *Proximal Policy Optimization* (*PPO*), introduced by Schulman et al. [13], is a policy-gradient *RL* algorithm

that improves the policy in small increments. It does this by comparing the probability of each action under the new policy to its probability under the old policy and ”clipping” that ratio, so it cannot change too much at once.

Because some batching actions may be invalid (for example, selecting a batch size below a minimum), we use *action masking* to prevent the agent from choosing those options. By setting the probability of invalid actions to zero, the agent focuses its learning on feasible decisions without modifying the core PPO update rule [14].

Details on how *PPO* is applied in our approach are provided in section 4.3.2.

2.5 Architecture Patterns

A common use case for *BPO*, also supported by *Optimos v2*, is to modify an existing simulation model, e.g. changing the timing condition of an existing batching policy rule. As those simulation parameters, often differentiated per activity or resource, can become quite complex, a log of every modification done by the *BPO* software is essential. Such a modification log allows process analysts to first understand the changes and then apply changes consistently to the real world process, or replay optimizations across different *BPs* without going through the optimization again. This log, allowing an *audit* of the modifications made, is consequently called an *audit log*.

One software architecture pattern that is often chosen for its audit capability is *Event Sourcing* (*ES*) [15]. First informally defined by Fowler [16], *Event Sourcing* describes a different approach to persisting application state of software system: Instead of saving the current snapshot of the program state in a relational database and mutating it over the programs' lifecycle, the current state is defined by a log of immutable events that, when applied on a base state, can be used to (re-)construct the current state.

Many systems built with *ES* also employ the *Command Query Responsibility Segregation* (*CQRS*) architecture [15]. On a high level, it proposes that two separate services should be used: one for querying *(reading)* the state and one for writing it. In combination with *ES*, *queries* access the event log in a read-only manner, while *commands* will produce new events [17, 18]. The *commands* are treated as first-class-citizen objects as defined by Gamma et al. [19].

Kabbedijk et al. [20] further elicited among others, three useful patterns to be used in conjunction with *CQRS*: First of all *Event Store*s are components that collect and persist *event logs*. *Aggregate Roots* provide a unified interface for similar, related data, e.g. one interface to access an invoices recipient and order items, even though, in the background, they may be implemented using

different classes and stored differently. Finally, *Command Handler*s implement logic to check if an action (or *command*) is valid based on the current state (often referred to as an *Accept* function [15]), and then possibly create a change event which gets appended to the other changes in the *event store*. Notably, since events in *ES* are immutable, those *commands* should be implemented as *pure functions*.

In addition to the previously discussed patterns, three more proved useful in the implementation of this thesis. The *Iterator* pattern, originally designed to abstract sequential list access [19], can also be used to abstract any iterative process – such as an optimization process – and grants control of the iteration to the *Iterators*'s user and yields an intermediate result at each step. The *Facade* pattern, combines a set of interfaces into a unified, high-level interface [19]. This is especially useful when interacting with a complex external library. Lastly, using the *Ghost* pattern, an object exists in a partial state, until its full state is explicitly requested, which is then loaded on demand [21]. This allows an object to be used and passed around with minimal memory overhead, and only consume significant resources when actually needed.

For an overview on how these architectures and patterns are implemented in *Optimos v2* refer to section 4.4.

3. Related Work

Liu and Hu [22] propose to group activities in a process into *Batch Processing Areas* (*BPA*). Activity instances that reach a *BPA* are accumulated until the batch size reaches a threshold. The authors suggest this threshold may be optimized via simulation but do not describe a method to do so. Pufahl et al. [23] adopt a similar model based on *batch regions*. In their model, the batch activation conditions depend both on batch size and time since batch creation (time-to-live timeout). Similarly, Natschläger et al. [24] consider batching policies based on batch size, time-to-live timeouts, and inactivity timeouts. Pufahl and Karastoyanova [25] propose a flexible batch execution model that, in addition to supporting batch activation rules, allows batches to be triggered based on contextual conditions or external triggers. However, none of the above studies deals with optimizing the batch activation rules.

Pflug and Rinderle-Ma [26] propose a model wherein instances of an activity are classified at runtime based on their data attributes. Activity instances belonging to the same class are processed together in batches whenever a resource becomes available. This approach does not seek to optimize the batching policy.

Pufahl et al. [27] outline an approach to optimize the batch size threshold w.r.t. a cost function that combines the ‘‘cost of waiting’’ plus the processing effort. The cost of using a given batch size threshold is estimated via queuing theory. The approach uses a grid (exhaustive) search, which is only practical for small search spaces. This approach does not consider batch activation rules that depend on time bounds (only batch size), and it optimizes each batch activity separately. In the presence of multiple batch activities in a process, the batch activation rule of an activity affects the time when cases reach other downstream activities. Hence, optimizing the activation rules of each activity separately may lead to suboptimal results. Accordingly, in our work, we optimize the batch activation rules of all batched activities in a process simultaneously.

Pufahl and Weske [2] propose a batch processing model unifying prior approaches. In this model, batch activation rules include three types of conditions: (1) batch size, (2) time-to-live or inactivity timeout, and (3) circadian time, e.g., a batch activates on Monday at 8:00 AM following its creation time. We adopt this latter approach and propose a method to find optimal activation rules w.r.t., waiting time, processing effort, and cost. Besides a batch activation rule, a *batching policy* includes other parameters, notably, the *execution order* of instances in a batch (parallel or sequential). This parameter determines the processing time of a batch: If parallel, the time taken by the longest activity instances determines the processing time of the batch,

while if sequential, the processing time of the batch is the sum of the processing times of its activity instances, minus a *discount factor*, as executing instances in batches is typically faster than processing them separately.

Wen et al. [28] present a discrete particle swarm optimization approach to dynamically optimize batch grouping and scheduling in a single workflow, using generic random perturbations to explore feasible and infeasible solutions. Similar to this thesis, they use Pareto fronts, but unlike this work, they do not tailor heuristics to batch activation rules (size, timeouts, circadian triggers) nor optimize multiple batched activities end-to-end.

Other studies consider the related problem of optimizing batch sizes and batch-to-resource assignment in production systems consisting of multiple workstations [29–31]. These studies use queuing theory to estimate the waiting times generated by different batch sizes and meta-heuristics (e.g., genetic algorithms) to explore the search space. These studies, however, do not consider batch activation policies that also depend on timeouts and circadian activation time.

Regarding this thesis (meta-)heuristics architecture, López-Pintado et al. [3, 32] describe a similar approach to optimization, but this first version of *Optimos* solely focuses on the optimization of resource availability calendars, and does not optimize batching policies.

4. Approach

4.1 Batching Model

The Definitions 1- 3 formalize the batching model (a.k.a., batching policy) used in this thesis, adapting the concepts from [5, 33]. Specifically, we incorporate the batching cost concept and formalize the batching activation rules supported by our approach.

Definition 1 (General Notations)**.** *Let $\mathcal{A}$ be the set of activities in a business process, and let $\mathcal{I}_a$ denote the set of instances of each activity $a \in \mathcal{A}$, where each instance $i \in \mathcal{I}_a$ represents an execution of activity a. We define the following time-related functions for any instance i: $\tau_e(i)$ represents the **enablement time**, when the activity instance becomes available for execution, $\tau_s(i)$ the **start time**, when execution begins, and $\tau_c(i)$ the **completion time** when it finishes. Let $\mathcal{R}$ be the set of resources (human or otherwise) who can execute activity instances.*

Definition 2 (Activity Batching)**.** *A **batch** is a tuple $\mathcal{B} = (\mathcal{I}_a^{\mathcal{B}}, r, \tau_s^{\mathcal{B}}, \tau_c^{\mathcal{B}}, \beta, \kappa)$, where $\mathcal{I}_a^{\mathcal{B}} = \{i_1, i_2, \ldots, i_n\} \subseteq \mathcal{I}_a$ is the set of batched instances of activity a; $r \in \mathcal{R}$ is the resource executing the batch; $\tau_s^{\mathcal{B}}$ and $\tau_c^{\mathcal{B}}$ are the start and completion times of the batch, and β is the batching type. If β is **sequential**, instances execute one after another, i.e., $\tau_s(i_l) = \tau_c(i_{l-1})$, for all $i_l \in \mathcal{I}_a^{\mathcal{B}}$ with $l > 1$. If β is **parallel**, all instances start and end together, i.e., $\tau_s(i_j) = \tau_s(i_l)$ and $\tau_c(i_j) = \tau_c(i_l)$ for all $i_j, i_l \in \mathcal{I}_a^{\mathcal{B}}$. Each $i_l \in \mathcal{I}_a^{\mathcal{B}}$ may belong to a different process case but follows the same batch constraints. The batch execution cost is $\kappa = \kappa_f + \kappa_v(|\mathcal{I}_a^{\mathcal{B}}|) + \kappa_r(r, \tau_c^{\mathcal{B}} - \tau_s^{\mathcal{B}})$, where κ_f is a fixed execution cost, κ_v depends on batch size, and κ_r depends on resource r and execution time $\tau_c^{\mathcal{B}} - \tau_s^{\mathcal{B}}$. We use the notation $\mathcal{B}_a^{\mathcal{P}}$ to denote the set of all batches of a given activity a executed within a business process execution $\mathcal{P}$.*

Definition 3. *A **batch activation rule** is a boolean function that determines when a batch $\mathcal{B} = (\mathcal{I}_a^{\mathcal{B}}, r, \tau_s^{\mathcal{B}}, \tau_c^{\mathcal{B}}, \beta, \kappa)$ is started, defined as follows:*

$$\alpha(\sigma_t) = \bigvee_{j=1}^{m} \Big(\bigwedge_{i \in G_j} c^i(\sigma_t) \Big), \tag{1}$$

*where σ_t is the **process state** at time t and $c_i(\sigma_t)$ are activation conditions. Each conjunction ($\wedge$) ensures that all activation conditions in a set G_j must be satisfied simultaneously, while the disjunction ($\vee$) allows any set G_j to trigger the batch activation. The activation conditions supported in this model include the following:*

- ***Size-based activation***, $c^{size}(\sigma_t)$*: The batch starts if the number of enabled activity instances waiting reaches a threshold* θ_v*, i.e.,*

$$c^{size}(\sigma_t) \equiv |\mathcal{I}_a^{\mathcal{B}}| \geq \theta_v. \tag{2}$$

- ***(Waiting) Time-based activation***:
  - ***Since first (time-to-live)***, $c^{wt\text{-}first}(\sigma_t)$*: The earliest activity instance waiting has waited at least* θ_f *time units, i.e.*

$$c^{wt\text{-}first}(\sigma_t) \equiv t - \tau_e(i_1) \geq \theta_f. \tag{3}$$

  - ***Since last (inactivity)***, $c^{wt\text{-}last}(\sigma_t)$*: The latest activity instance waiting has waited at least* θ_l *time units, i.e.,*

$$c^{wt\text{-}last}(\sigma_t) \equiv t - \tau_e(i_n) \geq \theta_l. \tag{4}$$

- ***Schedule-based activation***:
  - ***Daily hour rule***, $c^{hour}(\sigma_t)$*: The batch starts if the hour extracted from* σ_t *belongs to a predefined set of activation hours* $T_{allowed} \subseteq \{0, 1, \ldots, 23\}$*,*

$$c^{hour}(\sigma_t) \equiv hour(\sigma_t) \in T_{allowed}. \tag{5}$$

  - ***Day of the week rule***, $c^{day}(\sigma_t)$*: The batch starts if the weekday extracted from* σ_t *belongs to a predefined set of allowed days* $D_{allowed} \subseteq \{Monday, \ldots, Sunday\}$*,*

$$c^{day}(\sigma_t) \equiv weekday(\sigma_t) \in D_{allowed}. \tag{6}$$

Enabled activity instances accumulate in $\mathcal{I}_a^{\mathcal{B}}$ *until an activation rule is met. When* $\alpha(\sigma_t) = \texttt{true}$*, all the activity instances in* $\mathcal{I}_a^{\mathcal{B}}$ *are executed as a batch, with* τ_s *as the current time in* σ_t*, and* τ_c *determined by the batch type* β*. If the process execution ends before activation, any remaining instances are batched.*

Scheduled activation rules can combine $c^{\text{day}}(\sigma_t)$ and $c^{\text{hour}}(\sigma_t)$ to trigger batch activation at specific day-hour combinations, e.g., matching observed execution patterns:

$$c^{\text{sched}}(\sigma_t) \equiv \bigvee_{(d,h)\in\Omega} \Big(c^{\text{day}}(\sigma_t) = d \wedge c^{\text{hour}}(\sigma_t) = h\Big), \tag{7}$$

where Ω is the set of day-hour pairs defined by historical data or scheduling needs. For example, batches may start on Mondays at 3:00 PM or Thursdays at 2:00 PM to match high resource availability peaks. Also, activation rules like size-based could be designed considering previous executions of the process:

$$c^{\text{size}}(\sigma_t) \equiv |\mathcal{I}_a^{\mathcal{B}}| \geq \theta'_v, \tag{8a}$$

$$\text{with } \theta'_v = \lambda \cdot \frac{1}{|\mathcal{B}_a^{\mathcal{P}}|} \sum_{\mathcal{B} \in \mathcal{B}_a^{\mathcal{P}}} |\mathcal{I}_a^{B}|, \tag{8b}$$

where θ'_v is the allowed batch size threshold, calculated as the average batch size from past executions of activity a in process $\mathcal{P}$, scaled by λ. Here, **a)** if $\lambda > 1$, the batch size increases; conversely, **b)** if $0 < \lambda < 1$, the size decreases.

While batching may improve processing effort and reduce costs, rigid policies that do not consider process-specific characteristics can cause inefficiencies. In the following, we explore such scenarios and discuss heuristic interventions.

4.2 Intervention Heuristics

Aligned with the primary goal of this thesis to optimize costs and times simultaneously, we identified 19 scenarios that can impact one or both dimensions, either directly or indirectly. These scenarios are organized into four categories: waiting time, processing time, cost efficiency, and resource utilization. Each category targets a specific aspect of batch processing that affects cost or time. Then, the proposed heuristic interventions adjust the batching policy to address these scenarios, aiming to enhance the overall process performance. Additionally, we applied each scenario, in a simplified manner, to the example *BP* defined in section 2.2.2.

Heuristic Group 1. – Waiting Time Related Scenarios

Scenario 1: *The first instance in a batch typically waits the longest before execution, making it the main contributor to the overall waiting time in the batch. If this first instance waits significantly longer than other instances, e.g., due to low arrival rates or scheduling constraints, the batching activation may be excessively delayed.* **Then:** *Add a time-based activation condition* $c^{\text{wt-first}}(\sigma_t)$*, equation* (2)*, with*

$$\theta'_f = \lambda \cdot \frac{1}{|\mathcal{B}_a^{\mathcal{P}}|} \sum_{\mathcal{B} \in \mathcal{B}_a^{\mathcal{P}}} \max_{i_k \in \mathcal{I}_a^{B}} (t - \tau_e(i_k)).$$

This formula adjusts the batch activation threshold θ_f *by averaging the longest waiting times from previous batches of activity* a *observed during the process execution. It scales the result by*

λ to control the threshold's sensitivity to observed delays, e.g., $\lambda = 0.9$ lowers the threshold by 10%, allowing earlier activation.

Example: *Assume the* first *blood sample of a test run waits on average 100min,* **THEN:** *force a test to start if the first sample has at least waited for $100min \times 90\% = 90min$.*

SCENARIO 2: *The last instance in a batch typically waits the least before execution. However, its waiting time can become a bottleneck if the batch is close to reaching the size threshold and the next activity instance is unlikely to arrive soon due to resource unavailability or scheduling constraints. Thus delaying the execution of the entire batch.* **THEN:** *Add the time-based activation condition $c^{\textit{wt-last}}(\sigma_t)$, equation (4), with*

$$\theta_l' = \lambda \cdot \frac{1}{|\mathcal{B}_a^{\mathcal{P}}|} \sum_{\mathcal{B} \in \mathcal{B}_a^{\mathcal{P}}} \min_{i_k \in \mathcal{I}_a^B} (t - \tau_e(i_k)).$$

Similar to Scenario 1, this rule targets the last instance instead, adjusting the threshold based on the minimum waiting time:

Example: *Assume the* last *blood sample of a test run waits on average 100min,* **THEN:** *force a test to start if the last sample has at least waited for $100min \times 90\% = 90min$.*

SCENARIO 3: *When activities are frequently enabled or executed at specific times, but batches are misaligned and triggered at unscheduled intervals outside those timeframes, instances might accumulate, resulting in higher waiting times.* **THEN:** *Add a scheduled activation condition, $c^{\textit{sched}}(\sigma_t)$, equation (7), where Ω is the set of high-frequency day-hour pairs identified from historical data. These pairs correspond to execution peaks, scheduling batch activation during periods of high enablement or execution frequencies.*

Example: *Assume that doctors usually collect their samples in the morning, meaning at around 10:00 there is a peak of samples to be processed,* **THEN:** *add a new policy, forcing a batch to start at 10:00 each day.*

SCENARIO 4: *If batches are triggered when few resources are available, instances may wait longer for execution due to resource unavailability, thus increasing waiting times.* **THEN:** *Add an activation condition, $c^{\textit{sched}}(\sigma_t)$, equation (7), with a focus on resource availability. Thus, Ω is the set of day-hour pairs with the highest number of resources available, determined by analyzing the availability patterns of resources observed executing activity instances. Thus reducing the likelihood of delays due to resource unavailability.*

Example: *Assume that each Wednesday at 15:00* Lab Technician *(*LT*) shifts overlap, thereby providing an opportunity to prepare more samples at the same time,* **Then:** *force a batch to start Wednesdays at 15:00.*

Scenario 5: *Large batch sizes require more activity instances to wait before activation, potentially delaying execution and increasing waiting times.* **Then:** *Update an existing policy $c^{size} \in G$ with a new reduced threshold θ'_v as by equation* (8b).

Example: *Assume the current batching policy is to fill up the machine with 10 samples,* **Then:** *reduce the min. batch size to 9.*

Heuristic Group 2. – Processing Time Related Scenarios

Scenario 6: *For tasks with high processing times that can be executed in parallel, small batch sizes limit the opportunity for concurrent execution, increasing cumulative processing times, which could be reduced with larger batches.* **Then:** *Increase batch size threshold θ'_v of an existing policy $c^{size} \in G$ as by equation* (8b)*, to reduce the cumulative processing time via parallel execution.*

Example: *Assume that the current batch policy will start a test run with only 2 samples,* **Then:** *increase the min. samples required to 3.*

Scenario 7: *When tasks in a batch are executed sequentially or with low concurrency, increasing the batch size does not (significantly) reduce cumulative processing times, making batching no more efficient than individual activity execution in terms of processing times.* **Then:** *Update an existing policy $c^{size} \in G$ with a new reduced threshold θ'_v as by equation* (8b)*, because batching provides no advantage.*

Example: *Assume the current amount of samples required to start a batch is 11; 1 more than fits into the machine. That means per batch you'll need 2 sequential test runs,* **Then:** *reduce the min. batch size to 10.*

Scenario 8: *If batches are triggered close to periods of resource unavailability, activity instances may experience idle times when the resource becomes unavailable mid-execution, increasing overall processing times.* **Then:** *Add the scheduled activation condition, $c^{sched}(\sigma_t)$, equation* (7)*, where Ω is the set of day-hour pairs selected based on one of the following criteria:*

- *Pairs are chosen to enforce the batch starting at the beginning of a longer resource availability window period, according to their working calendars, to minimize interruptions.*

- *Pairs are selected for the batch to start on time slots from where the average processing time is less than or equal to the resource availability period, so the batch completes before resource unavailability.*

Example: *Assume a test run is commonly started at 17:45, 15min before lab closing time, then all samples in the batch will need to idle over the night, before the results can be processed,* **Then:** *trigger a batch already at 16:00.*

Scenario 9: *If batch activation occurs without aligning the batch waiting times to resource availability periods, activity instances may start execution during fragmented or short availability windows, causing interruptions and increasing idle times.* **Then:** *Adjust the time-based activation to align batch execution with optimal processing windows by updating existing $c^{\textit{wt-first}} \in G$ or $c^{\textit{wt-last}} \in G$, equations* (3) *and* (4)*, where the updated waiting time thresholds are:*

$$\theta'_f = \lambda \cdot \frac{1}{|\mathcal{B}_a^{\mathcal{P}}|} \sum_{\mathcal{B} \in \mathcal{B}_a^{\mathcal{P}}} \Delta_w(i_1), \quad \theta'_l = \lambda \cdot \frac{1}{|\mathcal{B}_a^{\mathcal{P}}|} \sum_{\mathcal{B} \in \mathcal{B}_a^{\mathcal{P}}} \Delta_w(i_n),$$

where: $\Delta_w(i_1)$ and $\Delta_w(i_n)$ are calculated by comparing the currently observed waiting times with the start of the most suitable execution window for the batch. This suitability is determined by the resources' availability and the estimated processing time of the batch, aiming to minimize idle times. Then, λ is a scaling factor for adjusting the thresholds.

Example: *Assume that* LT *make a break every four hours, which, if not well-timed, might interrupt running tests causing idle time,* **Then:** *start a batch after 3 hours of waiting time, so it can finish the test before a break occurs.*

Heuristic Group 3. – Cost Related Scenarios

Scenario 10: *When batch costs decrease with larger batch sizes, grouping more instances reduces the total cost compared to processing them separately, i.e., analogous to bulk purchasing, where buying in larger quantities reduces the cost per unit.* **Then:** *Update an existing policy $c^{\textit{size}} \in G$ with a new increased threshold θ'_v as by equation* (8b)*, because larger batches can reduce the overall process costs.*

Example: *Assume that transporting a non-full container of blood samples required packaging material (e.g. single use styrofoam) to stop samples from moving around, while a full(er) container does not require as much styrofoam,* **THEN:** *increase the min. amount of samples per transport (reducing styrofoam cost)*

SCENARIO 11: *High-cost activities might contribute the most to total process costs. Grouping these costly instances into larger batches can amortize shared fixed and variable costs, reducing the average cost per instance and lowering their impact on overall cost.* **THEN:** *Update an existing policy $c^{size} \in G$ with a new increased threshold θ'_v as by equation* (8b)*, to amortize shared costs of costly activity instances.*

Example: *Assume that the wear & tear of the machine running the test is by far the largest cost factor,* **THEN:** *increase the min. amount of samples to run per test, to decrease the wear & tear cost per tested sample.*

SCENARIO 12: *Frequently executed activities may contribute significantly to total process costs due to repeated fixed and variable costs. Grouping these recurring instances into larger batches may amortize them and minimize their impact on overall process costs.* **THEN:** *Update an existing policy $c^{size} \in G$ with a new increased threshold θ'_v as by equation* (8b)*, to amortize costs of very frequent activities.*

Example: *Because sometimes tests need to be rerun, the test activity is on average executed more than once per case making it the most frequent activity,* **THEN:** *increase its batch size first, potentially having a higher impact then e.g. the transport of the sample, as that will at most happens once per case.*

SCENARIO 13: *Low or mid-cost activities with similar enablement or execution patterns might often be overlooked for batching due to their low contribution to the overall cost. When executed at similar times, grouping these instances into larger batches synchronizes execution, minimizing logistics overhead and thus may improve cost efficiency.* **THEN:** *Update an existing policy $c^{size} \in G$ with a new increased threshold θ'_v as by equation* (8b)*, for low to mid-cost activity instances with similar execution patterns.*

Example: *Assume transport is much cheaper (per sample) then the cost of running the test (e.g. because machine wear & tear, highly paid* LTs*), but still often samples are collected at similar times (e.g. in the morning)* **THEN:** *increasing the batch size of the*

transport still makes sense, as over many cases the saved cost still is significant, while the cost to batch is also quite low.

Scenario 14: *Low-cost and low-frequency activities without synchronized patterns might have minimal impact on total process cost. Large batch sizes increase waiting times without saving costs.* **Then:** *Update an existing policy $c^{size} \in G$ with a new reduced threshold θ'_v as by equation* (8b)*, because smaller batches do not affect cost efficiency.*

Example: *Assume that samples that need to be stored for further tests, will only be stored in batches of 10 samples, but also because most patients are not actually ill, positive tests only sometimes happen at irregular intervals,* **Then:** *decrease the batch size for storage.*

Scenario 15: *When increasing batch size does not reduce cost, it may only introduce delays without improving cost efficiency. For example, when larger batches do not reduce fixed and variable costs due to set up costs remaining constant regardless of batch size.* **Then:** *Update an existing policy $c^{size} \in G$ with a new reduced threshold θ'_v as by equation* (8b)*, because smaller batches do not affect cost efficiency.*

Example: *Assume that doctors are informed of the test results via email, which a written in batches,* **Then:** *reduce the batch size because sending one email is exactly as expensive as sending multiple.*

Heuristic Group 4. – Resource Related Scenarios

Scenario 16: *High resource utilization above a certain threshold can create bottlenecks, causing delays due to resource unavailability or competing activity assignments. Larger batch sizes may hold resources for extended periods, thus increasing congestion.* **Then:** *Update an existing policy $c^{size} \in G$ with a new reduced threshold θ'_v as by equation* (8b)*, to lower resource congestion and minimize waiting times.*

Example: *Assume a large batch size of samples per test, also assume that preparing a sample for a test takes relatively long, meaning the* LT *will spend all its time preparing many samples for relatively few test runs, and thus will not often have the opportunity to e.g. send a result to a doctor,***Then:** *reducing the batch size a bit, so the* LT *does not need to prepare so many samples in one go, and therefore send doctor reports in between more often.*

Scenario 17: *Low resource utilization below a certain threshold may indicate underutilization, where resources are idle for long periods, increasing fixed costs relative to unproductive resources. Smaller batch sizes may fail to allocate available resources, reducing cost efficiency.* **Then:** *Update an existing policy* $c^{size} \in G$ *with a new increased threshold* θ'_v *as by equation* (8b)*, to better utilize available resource capacity.*

Example: *Opposite to Scenario 16, assume that tests are run very frequently (small batch size) and preparing a sample is quite fast, this means the* LT *will often have nothing to work,* **Then:** *increase min. required samples per test, thereby collecting a cost benefit without actually introducing any congestion.*

Scenario 18: *Activities with high variability in resource allocation require frequent switching between resources, increasing setup and transition times. Small batch sizes may introduce inefficiencies by frequently reallocating resources.* **Then:** *Update an existing policy* $c^{size} \in G$ *with a new increased threshold* θ'_v *as by equation* (8b)*, to reduce allocation switches and minimize overhead on transition times.*

Example: *Assume because of shifts one* LT *usually only has time to do a single test run before, after a hand-off break, a different Technician takes over,* **Then:** *increase the batch size, so the amount of samples processed in the limited time is maximized.*

Scenario 19: *Activities with low variability in resource allocation require fewer switches between resources. Large batch sizes may unnecessarily hold resources for extended periods, decreasing overall efficiency.* **Then:** *Update an existing policy* $c^{size} \in G$ *with a new reduced threshold* θ'_v *as by equation* (8b)*, to release resources sooner, improving allocation flexibility and reducing idle times.*

Example: *Assume that only the senior* LT *is allowed to inform doctors of the results, meaning if the senior* LT *is not available due to preparing a large amount of samples, no reports will be sent,* **Then:** *reduce the batch size to hold the Senior* LT *a shorter time.*

4.3 Optimization Description

Optimizing batching policies to minimize time and costs in a business process simultaneously requires dynamically adding, updating, or removing activation conditions. However, as discussed in section 2.4, the search space of activation condition candidates is infinite as time thresholds can take continuous values, and activation rules can combine in numerous ways. The heuristic interventions discussed in section 4.2 address this challenge by reducing the search space to

the most relevant scenarios, structuring the search process around relevant combinations, and prioritizing adjustments with potentially the highest impact on cycle time or cost.

The optimization should also balance the inherent trade-offs between cycle time and cost. Smaller batch sizes reduce waiting times but increase costs, as more instances may add a fixed cost independently. Conversely, larger batch sizes reduce costs by amortizing fixed costs across multiple instances but increase times as instances must wait longer to accumulate before triggering the batch. These conflicting objectives prevent finding a single solution that optimizes both dimensions simultaneously. Instead, they require a Pareto front, representing the set of non-dominated solutions where no other option is simultaneously better in cycle time and cost. A solution is on the Pareto front if improving one objective worsens the other. For example, smaller batches achieve a 2-day cycle time for \$500, and larger batches reduce costs to \$300 but increase the cycle time to 5 days. Both solutions are optimal as neither is better in both dimensions. This multi-objective optimization problem is formalized by definition 4.

Definition 4. *To minimize the average cycle time and cost, the goal is to find the optimal batching policy for each activity $a \in \mathcal{A}$. This is formulated as a multi-objective optimization problem, where the batching policy for each activity specifies the activation rules and batching characteristics as defined in definitions 2 and 3. The optimization problem is* $\min \frac{1}{D(\mathcal{B})} \{F_1(\mathcal{B}), F_2(\mathcal{B})\}$, *with*

$$F_1(\mathcal{B}) = \sum_{a \in \mathcal{A}} \sum_{\mathcal{B} \in \mathcal{B}_a^{\mathcal{P}}} \left(\tau_c^{\mathcal{B}} - \min_{i \in \mathcal{I}_a^{\mathcal{B}}} \tau_e(i) \right), F_2(\mathcal{B}) = \sum_{a \in \mathcal{A}} \sum_{\mathcal{B} \in \mathcal{B}_a^{\mathcal{P}}} \kappa, D(\mathcal{B}) = \sum_{a \in \mathcal{A}} \sum_{\mathcal{B} \in \mathcal{B}_a^{\mathcal{P}}} |\mathcal{I}_a^{\mathcal{B}}|,$$

where $F_1(\mathcal{B})$ is the cumulative cycle time, calculated as the sum of cycle times for all batches, from the earliest enablement of any instance in the batch to the completion of the batch, and $F_2(\mathcal{B})$ is the cumulative cost, calculated as the sum of all costs per batch. These values are then normalized by $D(\mathcal{B})$, i.e., the total number of activity instances executed, to minimize the average cycle time and cost per activity. Activity instances executed independently are treated as batches of size 1, triggered as soon as the allocated resource becomes available.

We follow an iterative data-driven approach to achieve the optimization goals stated in definition 4. Specifically, we consider event logs representing the execution of the business process subject to optimization. The approach automatically analyzes these logs to extract sources of inefficiencies, supported by the 19 scenarios described by the heuristic in section 4.2. For example, waiting time-related scenarios target the activity with high waiting times, ignoring those without delays. Then, they propose actions to reduce it based on different conditions. So, **Scenarios 1-2** require

delays being batching-related since updating a batching waiting threshold is meaningless on not batched activities. **Scenario 3** calculates the most frequent activity enablement time slots to align the execution to these times. Similarly, **Scenario 4** analyzes the start times of resources assigned to the most delayed tasks, detecting intersections in resource availability to synchronize the execution accordingly. For the sake of conciseness, we omit the detailed description of how each of the 19 scenarios is automatically discovered and analyzed from the event log. Readers can refer to the source code repository for the complete algorithms.

The iterative approach uses simulation to evaluate the impact of each action on cycle time and cost. If an event log of the process execution exists, the approach discovers a simulation model from it; otherwise, a simulation model must be provided. The following sections describe how this is achieved using hill-climbing, simulated annealing, and reinforcement learning meta-heuristics.

4.3.1 Hill-Climbing and Simulated Annealing Optimization

Hill-climbing (HC) and simulated annealing (SA) follow the iterative structure described in Algorithm 1, where each iteration applies heuristic interventions, evaluates the updated batching policies through simulation, refines the Pareto front and select potential candidates, which are perturbated to generate new policies heuristically (lines 2-17). The approach is configurable, allowing the selection of the most problematic scenario, a subset of the most problematic scenarios, or all detected scenarios for intervention (line 11). Since heuristic interventions are independent, multiple scenarios can be executed in parallel to improve computational efficiency. Despite sharing common steps, their search behavior and acceptance criteria differ as described in the following.

Hill-climbing (lines 6-7, 15–20) is a local search method that evaluates candidates within a fixed (small) radius from the Pareto front, using Euclidean distance (Algorithm 1, line 14). This radius mitigates errors from simulation stochasticity, preventing minor variations in the simulation from affecting the search. The algorithm accepts only candidates within this threshold (lines 15–19), incrementally improving solutions for fast convergence. However, restricting exploration to a local neighborhood increases the risk to stop at a local optimum.

In contrast, simulated annealing (lines 8-9, 20–26) is a global search strategy that stochastically accepts candidates based on a temperature-dependent probability distribution. Then, it selects them randomly from the perturbation queue, allowing exploration outside the local neighborhood. It starts with a high temperature, i.e., in the domain of real numbers, meaning it stochastically accepts solutions even if they perform worse, allowing it to explore the search space widely.

Algorithm 1 Unified Hill Climbing and Simulated Annealing Optimization

```
1: INPUTS: S ← SimulationModel, maxSol ← MaxSolutionsCount, alg ← SearchStrategy, radius ← ϵ (HillClimbing),
   temperature ← ∞, cF ← CoolingFactor (SimulatedAnnealing)
2: eLog, cTime, cost ← SimulateBProcess(S)              ▷ Simulate initial process for baseline performance
3: ParetoF ← { <S, cTime, cost> }                       ▷ Initialize Pareto front with the initial solution to optimize
4: Candidates ← InitializeQueue(alg)                    ▷ Sorted by distance to Pareto for HC, Random for SA
5: while sCount < maxSol ∧ IsNotEmpty(Candidates) do
6:    if alg is HillClimbing then
7:       <eLog, dist> ← PopBestCandidate(Candidates)                 ▷ Retrieve closest candidate
8:    else if alg is SimulatedAnnealing then
9:       <eLog, dist> ← PopRandomCandidate(Candidates)               ▷ Retrieve random candidate
10:   end if
11:   Scenarios ← ExtractConflictingScenarios(eLog)                  ▷ From heuristics in section 4.2
12:   for each sc in Scenarios do
13:      S' ← ApplyHeuristicAction(S, sc)                            ▷ Modify batch policy: actions in section 4.2
14:      eLog', cTime', cost' ← SimulateBProcess(S')                 ▷ Get updated policy performance
15:      dist ← GetEuclideanDistance(ParetoF, cTime', cost')         ▷ Dist to current Pareto
16:      if dist = 0 then                                            ▷ Non-dominated solution, add to Pareto front
17:         UpdateParetoFront(ParetoF, <S', cTime', cost'>)          ▷ Update optimal solutions
18:         EnqueueCandidate(Candidates, <eLog', 0>)                 ▷ Consider in future iterations
19:      else if alg is HillClimbing ∧ dist < radius then            ▷ Close to Pareto
20:         EnqueueCandidate(Candidates, <eLog', dist>)              ▷ Consider in future iterations
21:      else if alg is SimulatedAnnealing ∧ Random(0,1) < e^(−dist/temp) then
22:         EnqueueCandidate(Candidates, <eLog', dist>)              ▷ Candidate in temperature range
23:      end if
24:   end for
25:   if alg is SimulatedAnnealing then
26:      temp ← temp × cF                                            ▷ Reduce temperature to make search more strict
27:      DiscardOverLimit(Candidates, temp)                          ▷ Remove candidates out of temperature range
28:      if temp < ϵ then
29:         alg ← HillClimbing, radius ← 0                           ▷ Switch to HC, keep only optimal candidates
30:      end if
31:   end if
32: end while
33: return ParetoF                                                   ▷ Return Pareto-optimal solutions
```

As the temperature decreases, the chance of accepting worse solutions decreases. Eventually, when the temperature approaches zero, the algorithm behaves like hill-climbing, accepting only optimal candidates. This cooling schedule enables simulated annealing to escape local optima by widening the search space in the early stages while refining solutions in later iterations.

4.3.2 Reinforcement Learning for Batching Policies

The *Reinforcement Learning* (*RL*) approach (Algorithm 2) optimizes batching policies as a sequential decision-making problem, where an *RL* agent (a *PPO* agent in our implementation) iteratively refines policies by interacting with simulation models. The agent learns a policy over time, guided by a Markov Decision Process (MDP): the state represents the current process simulation, actions correspond to heuristic interventions (section 4.2), and rewards evaluate actions effectiveness. The goal is to maximize long-term rewards by iteratively selecting the best intervention strategies.

Algorithm 2 Reinforcement Learning-Based Optimization

```
1:  INPUTS:
        S ← SimulationModel, mI ← MaxIterations,
        RLModel ← ReinforcementLearningModel, Buffer ← ExperienceBuffer
2:  eLog, cTime, cost ← SimulateBusinessProcess(S)          ▷ Simulate initial process model
3:  ParetoF ← { <S, cTime, cost> }                           ▷ Initialize Pareto front
4:  State ← ExtractState(eLog, cTime, cost)                  ▷ Extract initial RL state representation
5:  for iteration = 1 to mI do
6:      Actions ← GetAvailableInterventions(S)               ▷ Retrieve possible heuristic interventions
7:      if Actions = ∅ then
8:          Break                                            ▷ Terminate if no valid actions exist
9:      end if
10:     ActionMask ← MaskUnavailableActions(Actions)         ▷ Mask invalid interventions
11:     α ← SelectAction(RLModel, Actions, ActionMask)       ▷ Predict best intervention
12:     S' ← ApplyHeuristicAction(S, α)                      ▷ Modify batching policy
13:     eLog', cTime', cost' ← SimulateBusinessProcess(S')   ▷ Evaluate new policy
14:     NextState ← ExtractState(eLog', cTime', cost')       ▷ Extract new state representation
15:     reward ← EvaluateImprovement(cTime, cost, cTime', cost')   ▷ Compute reward
16:     StoreTransition(State, α, reward, NextState, Buffer) ▷ Store experience
17:     if IsParetoOptimal(S', ParetoF) then                 ▷ Update Pareto front if non-dominated
18:         UpdateParetoFront(ParetoF, <S', cTime', cost'>)
19:     end if
20:     if ShouldTrain(Buffer) then                          ▷ Check training conditions
21:         TrainRLModel(RLModel, Buffer)                    ▷ Update RL model
22:         ClearExperienceBuffer(Buffer)                    ▷ Reset buffer
23:     end if
24:     State ← NextState                                    ▷ Update state for next iteration
25: end for
26: return ParetoF                                           ▷ Return Pareto-optimal solutions
```

In Algorithm 2, the *RL* agent initializes the simulation and extracts a state representation (lines 2–4). It then retrieves feasible heuristic interventions (lines 5–9) and predicts an action (line 10), modifying the batching policy and running a new simulation to assess its impact (lines 11–12). The transition is recorded, and a reward function evaluates effectiveness (lines 13–15). The reward function is defined as $r(S, S') = r_{\text{dom}}$ if S' dominates all solutions in the Pareto front, i.e., it is at least as good in all objectives and strictly better in at least one; r_{imp} if S' does not dominate but improves at least one Pareto-optimal solution; and r_{pen} otherwise, where $r_{\text{dom}} > r_{\text{imp}} > r_{\text{pen}}$, so that the agent prioritizes solutions enhancing the Pareto front while penalizing ineffective modifications. Non-dominated solutions are added to the Pareto front (lines 16–17). The *RL* model is trained incrementally using past experiences stored in a buffer (line 15). Once enough experience is accumulated, the policy is updated (lines 18–20), allowing continuous adaptation rather than relying on predefined search rules.

Unlike fixed meta-heuristics like hill-climbing or simulated annealing, the *RL* strategy dynamically balances exploration and exploitation. It explores broadly early in training and refines solutions over time but requires multiple iterations, making it sensitive to poor initialization. Also, training overhead can be significant in large-scale processes. Although *RL* refines the batching policy via a black-box model, the final output remains a white-box batching policy (Definitions 2-3).

4.4 Software Architecture

We use the *Python*[4] programming language for the implementation of *Optimos v2*. For its simulations it relies on the current version of the *Prosimos*[5] *BPS* engine designed by López-Pintado et al. [34]. The *reinforcement learning* component extends the *PPO* implementation by Huang and Ontañón [14] available as a *Python* module within the *Stable Baselines3 Contrib*[6] package. *Optimos v2* is first and foremost a *Command Line Interface* (*CLI*) program that can be run and configured without a graphical interface. However, a self-contained *GUI* was additionally developed described in more detail in chapter 6.

[4] https://www.python.org
[5] https://github.com/AutomatedProcessImprovement/Prosimos
[6] https://github.com/Stable-Baselines-Team/stable-baselines3-contrib

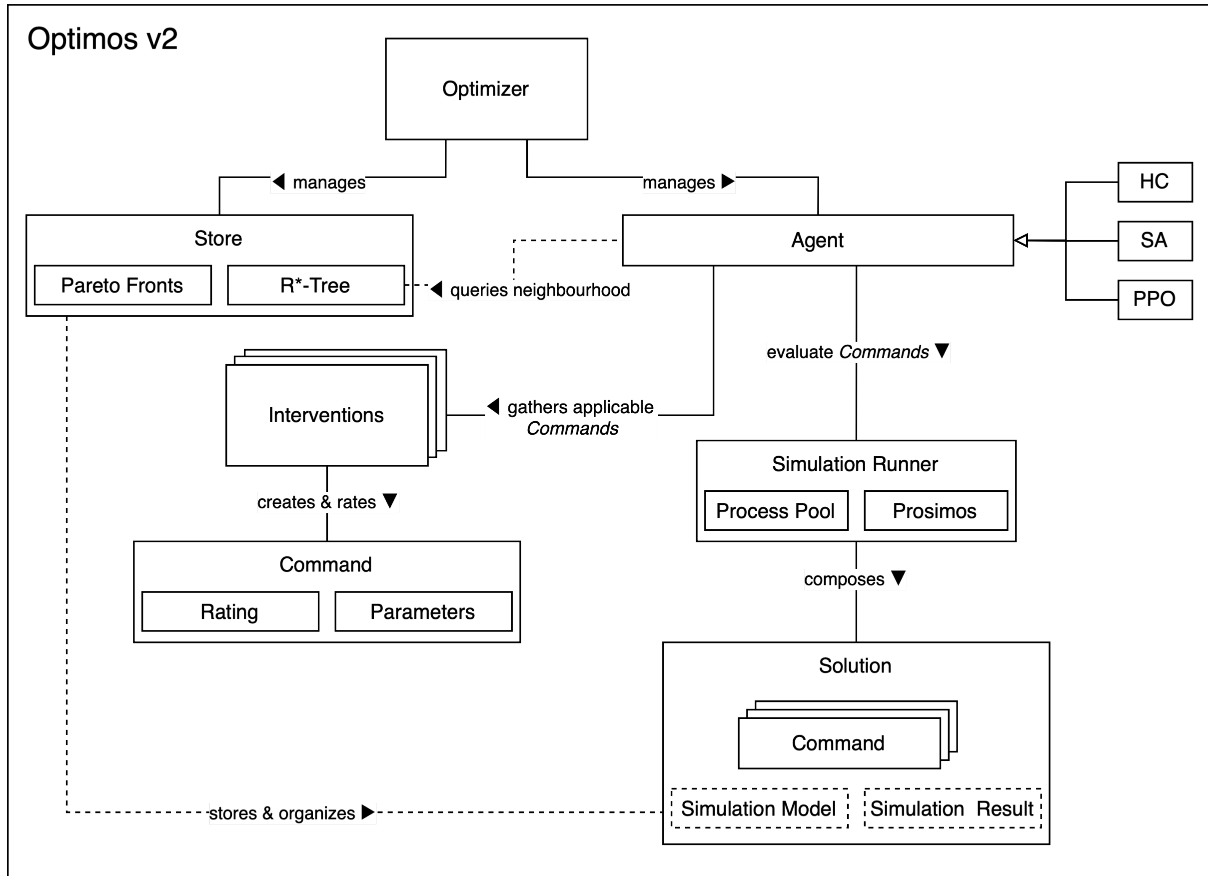


Figure 3. High-level overview of the *Optimos v2* architecture.

Figure 3 illustrates the *Event Sourcing*- and *Command Query Responsibility Segregation*-inspired architecture of *Optimos v2*. The most important components are the *Commands*, *Optimizer*, *Store*, *Agent*, *Simulation Runner*, and *Solution*. The *Optimizer* coordinates the iterative search process by delegating the selection of applicable *Commands* to the *Agent*, which implements a configurable optimization strategy. These *Commands* are then evaluated through *BPS* using the *Simulation Runner*. The resulting *Solutions* are stored and organized in the *Store*, which maintains the Pareto front and solution neighborhood. The following sections describe these components and their interactions in more detail.

Every heuristic intervention is modeled as a *command* – a pure function that produces a new state based on the current state and specific parameters. If these parameters are not valid (or the resulting state is invalid), the command will be rejected by the *agent's command handler*. Each intervention additionally defines its own *command creator* which proposes a *command* instance based on the current state (i.e., it selects suitable parameters for the intervention) and assigns this *command* instance a rating based on the likelihood of it improving the current state.

The *Agent* then selects which intervention *commands* to evaluate using *BPS*, which in turn generate new *solutions* that possibly serve as the base state for the next iteration. The agent is implemented by using the *Strategy Pattern*, which gives every implementation (*HC*, *SA* and *PPO*) the same interface, facilitating simple switching between agent implementations.

Moreover, *Agents* and *Interventions* follow common superclasses, with abstract factory methods. This design makes it straightforward to add new agents or interventions – essentially following a *plugin-in* architecture.

The *Simulation Runner* component acts as a *facade* to, providing a simplified interface for interacting with the *Prosimos BPS* engine and for managing parallel simulation of different simulation models. This component is also responsible for running multiple, concurrent simulations per *simulation model*, to obtain median values and reduce the impact of simulation variance

A *solution* is an *aggregate root* that encapsulates the *simulation model*, simulation results, and Pareto values via a unified interface. It is uniquely identified by the sequence of *commands* which generated it. To improve performance, it also acts as a *Memento* (as defined by Gamma et al. [19]), meaning the *simulation model* calculated by the series of *commands* is persisted (i.e.,*memorized*), allowing faster generation of subsequent simulations and reports.

Similarly, simulation results are persisted, as evaluating the modified policies by means of simulations represents the most computationally expensive step of the optimization process. A notable benefit of the immutable nature of *ES* events, as well as the pure *commands* of *CQRS*, is that parallel execution using multiple processes is straightforward to implement. *Optimos v2* leverages this to simulate multiple different interventions at the same time. To obtain thread-safe and performant hashes for uniquely identifying objects, i.e. *Commands*, we used *xxhash 3*[7].

The *Store* functions as the state repository (an *event store*), maintaining an append-only record of all immutable solutions and their associated metadata. It is responsible for managing the Pareto front and the entire evaluation neighborhood. For this, we use an R^*-Tree, a highly efficient data structure for storing and querying spatial data [35], which stores the Pareto values of all previously evaluated solutions. This enables efficient execution of common queries such as nearest-neighbor-lookup (PopBestCandidate, in algorithm 1 L7, for our *HC* implementation)

[7] https://xxhash.com/

or a random-neighbor selection within a given distance (PopRandomCandidate, in algorithm 1 L9, for *SA* implementation, where distance is determined by the *temperature*).

Orchestrating all parts is the *Optimizer*. It instantiates the *Agent* based on the configured meta-heuristic (*HC*, *SA*, *PPO*), manages termination criteria, and provides an *Iterator*-style interface for external tools. We use this interface for a real-time *TensorBoard* dashboard in the evaluation (see section 5.4), or to provide the *GUI* with live updates (see chapter 6).

Another (optional) feature is the to-disk archiving of the current state as well as single evaluations. This feature was extensively used during the evaluation and provides the user with a full solution objects, and regular resume-points, so long-running optimizations can be continued anytime. We used the *pickle*[8] library for this. For very complex simulation models, this mechanism can serve as the primary storage backend for simulation results and simulation models, following a *ghost* pattern to preserve memory.

The source code and further documentation for *Optimos v2* are available on GitHub[9].

---

[8] https://docs.python.org/3/library/pickle.html

[9] https://github.com/AutomatedProcessImprovement/optimos_v2

5. Evaluation & Testing

To evaluate our approach, or more precisely the quality of the Pareto front approximation achieved by it, we consider convergence (distance to the optimal Pareto front) and diversity (coverage of the solution space) [36]. A good approximation should be close to the optimal front while preserving structural diversity. Thus, we define two evaluation questions:

EQ1 How good are the Pareto fronts discovered by our proposal regarding convergence and diversity?

EQ2 How much gain does our approach achieve over the initial process?

5.1 Datasets and Experimental Setup

We relied on 10 real-life business processes with diverse characteristics to answer the evaluation questions. We discovered the simulation models to optimize from their corresponding event logs using the tool Simod[10]. To evaluate the impact of the heuristic interventions, we use the simulation tool Prosimos[11], supporting the Simod models. Table 1 summarizes the characteristics of these business processes, including the number of resources (RES), activities (ACT), and control-flow gateways —parallel (AND), exclusive (XOR), and inclusive (OR) —defining synchronization and decision points. Flow arcs (ARCS) illustrate structural complexity, with some processes having highly unstructured, spaghetti-like behavior. The SIM-Time column reports the average simulation time (in seconds) over ten runs, each executing 1,000 process instances.

The event logs used in our evaluation do not contain information on batching or the impact of batch size on execution. Also, they do not include cost-related data, information typically hidden from public records. Since costs usually relate to the actual execution of activities, we chose not to introduce arbitrary cost values.

Accordingly, we redefine the cost function κ_r based on processing times, which could be amortized through parallel execution but increasing waiting times: $\kappa_r = \lambda \sum_{i \in \mathcal{I}_a^B} p_i$ where p_i represents the processing time of instance i, and λ is a scaling factor. This formulation excludes (from p_i) idle periods due to resource unavailability during activity execution. We excluded processing time from cycle time to avoid redundancy in optimization objectives, considering

[10] https://github.com/AutomatedProcessImprovement/Simod
[11] https://github.com/AutomatedProcessImprovement/Prosimos

Table 1. Characteristics of the real-life business processes.

		RES	ACT	AND	XOR	OR	ARCS	SIM-Time
1	ACC	561	18	-	17	-	47	0.62
2	BP12	58	6	-	4	-	14	0.3
3	BP17	148	8	-	8	-	23	0.4
4	BP19	311	37	19	67	22	277	16.5
5	CALL	66	8	-	14	-	30	0.4
6	GOV	15	98	3	114	2	365	3.5
7	INS	125	11	-	13	-	33	0.8
8	PRD	48	26	38	70	-	293	0.5
9	SEP	25	16	4	22	7	82	1.5
10	TRF	29	11	4	12	4	52	1.0

only waiting, and idle times not captured by the cost function. Thus preserving the optimization problem (Definition 4) and ensuring the Pareto front balances execution efficiency and batching effectiveness (productive vs. unproductive time).

To analyze the impact of batching constraints on processing times, we adjust the scaling factor λ in the cost function $\kappa_r = \lambda \sum_{i \in \mathcal{I}_a^B} p_i$ we considered two scenarios. First, *Parallel batching* ($\lambda = \frac{1}{|\mathcal{I}_a^B|}$), where processing time is amortized across instances, so that batch execution time equals the longest individual activity. Second, *Hybrid batching* ($\lambda = \frac{0.5}{|\mathcal{I}_a^B|}$) balances parallel and sequential execution by scaling processing time under sequential execution by a 0.5 factor.

We evaluated these two execution scenarios, *parallel* and *hybrid*, using the meta-heuristics hill-climbing (HC), simulated annealing (SA), and reinforcement learning (RL). Our heuristic-guided variants, HC+, SA+, and RL+ (section 4.3), incorporate the proposed heuristics, while the baselines, HC-, SA-, and RL-, work without a heuristic guide, relying on the standard search mechanisms of each meta-heuristic via neighborhood random perturbations. Thus quantifying the impact of heuristic-driven optimization against standard meta-heuristic search.

5.2 Metrics

Since the actual Pareto front is unknown, we follow [37] to construct a reference Pareto front (`PRef`), which includes all non-dominated solutions from all algorithm runs. We define `PAprox`

as the Pareto front approximated by a single algorithm. To address the experimental questions, we evaluate three metrics:

- **EQ1- Convergence**: The Averaged Hausdorff distance [36] measures convergence as the mean root mean squared (RMS) distance between $\texttt{PAprox}$ and $\texttt{PRef}$: $\frac{1}{2}\sum_{S\in\{\texttt{PAprox},\texttt{PRef}\}}\sqrt{\frac{1}{|S|}\sum_{x\in S}\min_{y\in\bar{S}} d^2(x,y)}$, where S iterates over both sets and $\bar{S}$ is the opposite set. A lower value means a better convergence.

- **EQ1 - Diversity**: Purity [37] measures the proportion of $\texttt{PAprox}$ solutions included in $\texttt{PRef}$, given by: $|\texttt{PAprox} \cap \texttt{PRef}|/|\texttt{PAprox}|$. A higher purity indicates a better $\texttt{PAprox}$, with a maximum value of 1.

- **EQ2 - Gain**: Cycle time difference between the initial solution S_0 and $\texttt{PAprox}$ is given by: $\mathbb{E}[CT(S_0,n)] - \min\{\mathbb{E}[CT(S,n)] \mid S \in \texttt{PAprox}\}$, where $\mathbb{E}[CT(S,n)]$ is the mean cycle time per process case, computed as: $CT(S,n) = \max\{\tau(e) \mid e \in E_n\} - \min\{\tau(e) \mid e \in E_n\}$ with E_n as the set of activity instances in process case n and $\tau(e)$ the timestamp of activity instance e.

5.3 Results

Tables 2-4 present the results for convergence, diversity, and gain, highlighting cases where heuristic-driven approaches outperform all baselines (green), at least one baseline (blue), and underperform (or equal) all baselines (red).

Regarding **EQ1**, Table 2 presents convergence results using the averaged Hausdorff distance. Across parallel and hybrid settings, heuristic-guided approaches outperform at least one baseline in 85% of cases, confirming their impact on Pareto front convergence. RL+ is the best-performing method overall, surpassing RL- in 80% of cases and all baselines in 65%. HC+ follows with a 65% improvement rate over HC-, while SA+ achieves the lowest gains, surpassing SA- in only 45% of cases, indicating a weaker contribution of heuristics to SA's search process. The parallel scenario shows better convergence due to more flexibility in amortizing processing times, whereas hybrid execution introduces constraints that reduce the parallelization effects. Finally, columns labeled $++$ and $--$, which compare joint Pareto fronts from all heuristic-guided (++) versus non-guided (–) solutions, show that ++ achieves better convergence in 60% of cases, confirming that heuristic-driven solutions collectively improve convergence.

Continuing with **EQ1**, Table 3 presents Purity results, assessing the diversity and contribution of each approximated Pareto front to the reference optimal. Here, heuristic-guided approaches

Table 2. **EQ1 Convergence** – Averaged Hausdorff Distance Results.

	Parallel								**Hybrid (factor $\lambda = 0.5$)**							
	HC+	**SA+**	**RL+**	**HC-**	**SA-**	**RL-**	**++**	**–**	**HC+**	**SA+**	**RL+**	**HC-**	**SA-**	**RL-**	**++**	**–**
ACC	3.14	1.97	0.27	1.25	10.7	0.89	0.00	0.79	25.1	25.2	25.7	25.3	0.40	26.0	25.1	0.37
BP12	2.74	4.28	0.09	4.37	4.95	2.78	0.08	2.77	1.79	1.65	1.53	2.01	1.72	1.03	1.53	1.03
BP17	0.27	3.90	1.62	2.93	0.69	0.04	1.37	0.02	1.94	0.99	0.73	1.18	5.73	2.30	0.12	5.88
BP19	10.1	8.46	2.02	10.2	5.61	2.08	1.80	0.12	6.71	5.48	0.00	6.30	151.6	112.7	0.00	119.6
CALL	8.23	9.17	0.00	6.93	15.1	5.70	0.00	5.70	60.1	75.4	7.44	113.6	0.60	4.87	7.44	0.33
GOV	11.7	38.8	4.72	52.7	48.7	14.4	2.36	14.2	0.29	111.0	53.9	126.6	132.8	111.9	0.17	108.41
INS	5.80	6.99	6.19	6.43	0.30	6.20	5.79	0.23	2.75	2.48	0.35	1.42	0.60	0.72	0.08	0.56
PRD	9.05	12.3	1.09	12.3	2.55	5.53	0.34	3.32	26.9	40.3	3.20	39.0	39.4	16.4	0.00	15.9
SEP	12.7	13.0	5.21	16.8	5.13	16.4	3.66	5.25	57.5	155.4	102.7	177.8	3.11	384.8	95.2	3.11
TRF	30.8	30.3	3.84	30.2	30.7	8.51	3.83	8.51	65.8	69.1	63.1	31.9	26.9	13.1	63.1	6.08

Table 3. **EQ1 Diversity** – Purity Results.

	Parallel								**Hybrid (factor $\lambda = 0.5$)**							
	HC+	**SA+**	**RL+**	**HC-**	**SA-**	**RL-**	**++**	**–**	**HC+**	**SA+**	**RL+**	**HC-**	**SA-**	**RL-**	**++**	**–**
ACC	0.58	0.06	0.36	0.01	0.01	0.00	0.99	0.01	0.66	0.19	0.00	0.03	0.13	0.00	0.84	0.16
BP12	0.12	0.19	0.33	0.00	0.00	0.26	0.72	0.28	0.09	0.24	0.22	0.00	0.00	0.46	0.54	0.46
BP17	0.34	0.29	0.10	0.01	0.06	0.26	0.69	0.31	0.26	0.21	0.26	0.00	0.00	0.26	0.74	0.26
BP19	0.05	0.00	0.09	0.05	0.14	0.68	0.14	0.86	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00
CALL	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.36	0.00	0.21	0.43	0.36	0.64
GOV	0.00	0.35	0.29	0.00	0.06	0.29	0.65	0.35	0.00	0.85	0.00	0.00	0.08	0.08	0.85	0.15
INS	0.64	0.17	0.10	0.10	0.17	0.10	0.81	0.19	0.34	0.26	0.34	0.03	0.11	0.03	0.89	0.11
PRD	0.26	0.04	0.18	0.45	0.08	0.00	0.48	0.53	0.61	0.09	0.30	0.00	0.00	0.00	1.00	0.00
SEP	0.33	0.00	0.59	0.04	0.04	0.00	0.93	0.07	0.00	0.38	0.00	0.00	0.62	0.00	0.38	0.62
TRF	0.00	0.00	0.30	0.00	0.00	0.70	0.30	0.70	0.00	0.00	0.69	0.00	0.13	0.19	0.69	0.31

outperform at least one baseline in 70% of cases, showing stronger performance (given more significant differences in the metrics results) than convergence, indicating heuristics help explore a broader range of trade-offs even in cases when they do not find the closest solutions to the optimal front. HC+ achieves the highest diversity improvement individually, surpassing HC- in 55% of cases, tying in 40%, and underperforming in 5%. SA+ and RL+ each outperform their baselines (SA- and RL-) in 50% of cases, tie in 20%, and underperform in 30%. Additionally, when considering a joint Pareto front built from heuristic-guided solutions (++ vs. –), diversity improves in 75% of cases, confirming the impact of heuristics in expanding the solution space.

Finally, to answer **EQ2**, Table 4 shows cycle time improvement per process case, measured in hours. While the objective function optimizes individual waiting and idle times vs. processing time per activity, external constraints like multitasking and delays may affect cycle time per case. Despite this indirect impact, not modeled by the objective function, heuristic-guided

Table 4. **EQ2 Gain** – Cycle Time Improvement Results.

	Parallel								Hybrid (factor $\lambda = 0.5$)							
	HC+	**SA+**	**RL+**	**HC-**	**SA-**	**RL-**	**++**	**–**	**HC+**	**SA+**	**RL+**	**HC-**	**SA-**	**RL-**	**++**	**–**
ACC	0.70	1.26	0.00	0.00	0.68	0.00	1.26	0.68	1.09	1.07	-0.06	1.07	0.75	-0.06	1.29	1.07
BP12	-0.08	0.17	-0.13	-0.13	-0.13	-0.13	0.17	0.05	0.08	0.09	-0.12	0.55	-0.22	-0.03	0.08	0.55
BP17	0.00	0.00	0.00	0.00	0.00	0.00	-0.09	0.00	0.41	0.30	0.35	0.30	0.30	0.43	0.41	0.43
BP19	30.5	26.1	23.9	28.8	28.4	13.9	30.47	28.8	7.55	8.78	3.01	-11.4	-6.26	-5.38	3.01	-6.26
CALL	672	646	721	689	689	721	721	721	352	418	728	187	728	726	721	721
GOV	5.48	12.8	0.00	5.48	10.7	8.79	12.8	10.7	22.7	21.0	21.1	15.3	21.1	20.5	22.7	20.6
INS	0.00	0.19	-0.13	0.00	0.00	-0.13	0.19	-0.13	0.00	0.00	0.68	0.00	0.00	0.00	-0.58	0.00
PRD	1.15	1.15	1.15	1.15	1.78	0.00	0.84	1.78	1.75	0.55	-1.19	0.55	0.26	-1.19	1.75	0.26
SEP	4968	4863	4913	4964	4961	4915	4968	4963	3999	2090	3318	2261	3805	2414	3999	3805
TRF	760	770	877	772	826	901	877	901	451	400	760	799	864	738	760	864

approaches reduced cycle time in 60% of cases and kept similar times in 40% of the remaining cases from the initial process, showing their capacity to improve overall execution cycle times. Note that negative values in the table indicate cycle time increases due to batching. However, values between 0 and -1 are small enough to be attributed to simulation errors, meaning cycle time remains unchanged regarding the original process. Individually, HC+ shows the biggest improvement, beating HC- in 60% of cases, tying in 25%, and underperforming in 15%. RL+ and SA+ each outperform their baselines in 40% of cases, but RL+ ties in 35% and underperforms in 25%, while SA+ ties in 15% and underperforms in 45%.

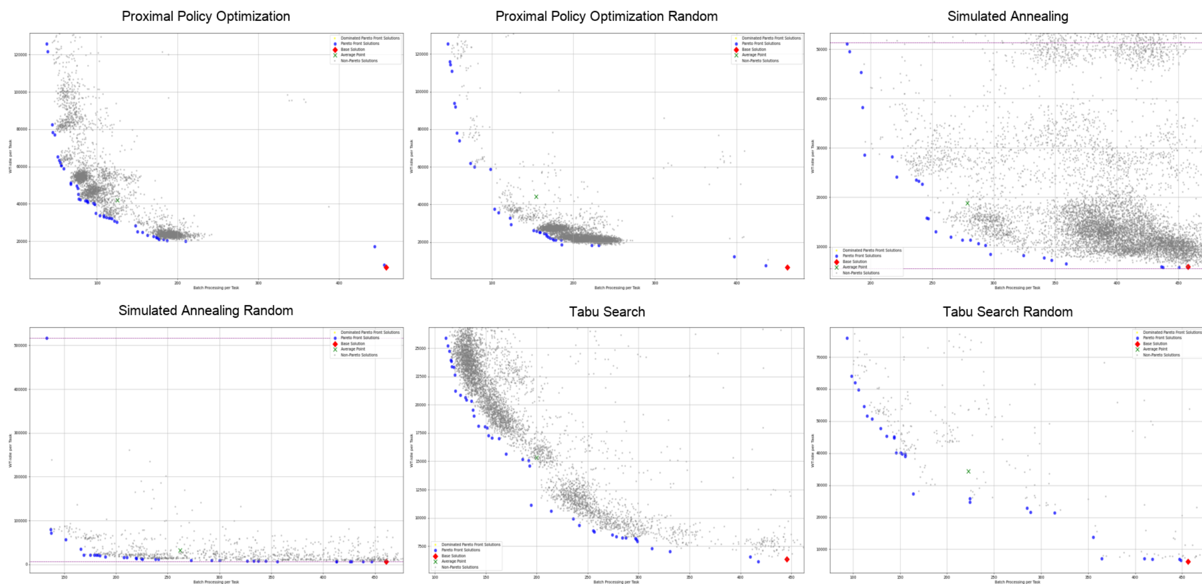


Figure 4. Example of Pareto fronts and non-optimal solutions (All algorithms, *ACC - Parallel*)

A more extensive evaluation confirming the findings and full event log descriptions, excluded for the sake of conciseness, can be accessed from the *Optimos v2* GitHub-Repository[12]. It includes computational overhead, solution counts, various convergence and spread metrics, graphical plots of Pareto fronts, such as in fig. 4, explored solutions, and metric progressions, among other insights.

5.4 Static Analysis & Testing

As the *Optimos v2* source code is relatively large comprising nearly 50,000 *Lines of Code* (*LOC*) (see table 5), it is essential to use automated tools to check code changes for correctness and best-practice conformance. Without such automation, it becomes impractical for developers to assess the side effects of changes manually. To address this, *Optimos v2* employs three complementary approaches.

Component	**Total Lines**	**Code**	**Comments**	**Blanks**
Core Optimizer	13 155	10 312	1 440	1 403
Core Optimizer Tests	8 318	6 180	873	1 265
Evaluation Tooling	14 175	10 845	1 624	1 706
Optimos *GUI* Service & Server	467	320	54	93
TypeScript Frontend	8 778	7 959	172	646
Configurations & misc.	4 442	3 395	159	888
All Components	49 335	39 011	4 322	6 001

Table 5. Project size by component (lines of code, comments, blanks).

First, *linting* with the *ruff*[13] analyzer. *Linting* is a static code analysis technique, where, without actually executing the code, common mistakes as well as style problems are flagged for the developer. *Optimos v2* uses over 250 different *ruff* linter checks, e.g. enforcing variable naming conventions or requiring documentation on public methods, in alignment with the widely adopted PEP-8 style guide [38]. Another tool that supports the developer is *cspell*[14], which statically checks code identifiers as well as documentation for correct English spelling.

[12] https://github.com/AutomatedProcessImprovement/optimos_v2
[13] https://github.com/astral-sh/ruff
[14] https://cspell.org

Secondly, we use *Python type hints*[39] and the *pyright*[15] static type-checker to ensure type correctness across the codebase. Type hints allow us to catch potential type-related bugs before runtime by explicitly declaring expected types for function parameters and return values. The type checker validates these annotations during development, helping prevent common programming errors like passing incompatible argument types or accessing undefined attributes.

Finally, to validate the core logic of the optimizer, we implemented 230 *unit* and *integration* tests based on the *pytest*[16] framework, achieving a *statement coverage* of 92%. Intervention integration-tests are written, so that *positive* and *negative* scenarios for each intervention are covered, i.e. scenarios where the heuristic intervention should apply, and those where it shouldn't. Core logic like the management of the *Pareto front* or the functions on the *r*-Tree*, are tested on a unit-test, per-method basis, so they can be relied upon.

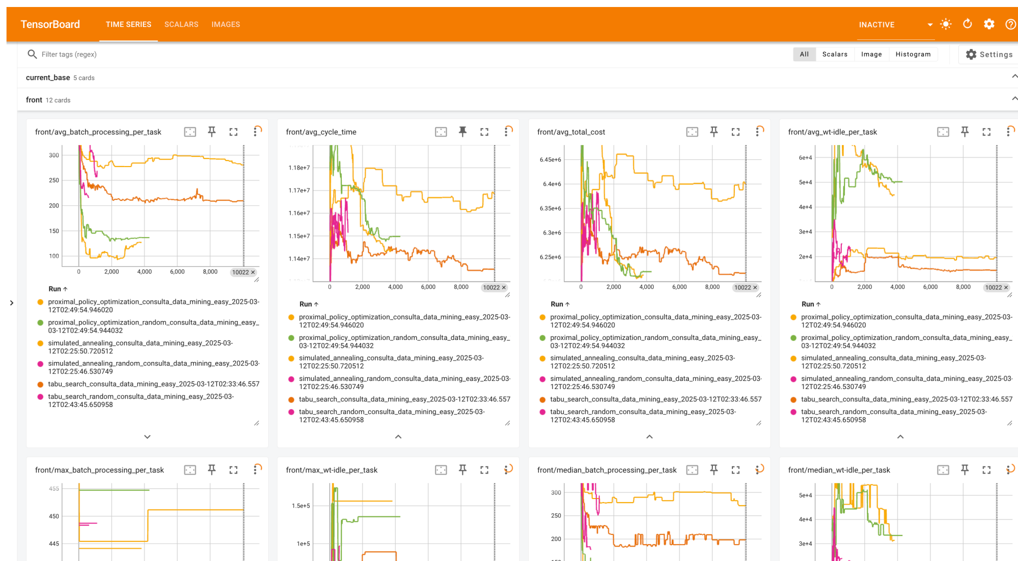


Figure 5. Screenshot of the *TensorBoard* showing real-time optimization metrics.

For manual testing, *Optimos v2* provides a configurable logging framework, based on *Pythons logging* module [17], that can be configured to show color-coded messages of different severities to the console or even write logs to file. Additionally, to monitor the optimization process itself, we implemented an interface to *TensorBoard*[18], to get a multitude of live statistics and visualization,

[15] https://microsoft.github.io/pyright/
[16] https://docs.pytest.org/en/stable/
[17] https://docs.python.org/3/library/logging.html
[18] https://www.tensorflow.org/tensorboard

even for a remotely running optimization. It can show different optimizations at the same time, see fig. 5 for an example. This needs to be differentiated from the *GUI* presented in chapter 6, as this dashboard is still very detailed, is read only and not for a non-technical user.

All static analysis and tests are integrated into a *Continuous Integration* (*CI*) pipeline to ensure consistent quality and prevent regressions.

Additionally, following the techniques described by Gorelick and Ozsvald [40] we also conducted manual runtime- and memory-analysis using the *VizTracer*[19] and *Pympler*[20], to ensure no memory leaks or other performance bottlenecks are present in the optimization logic of *Optimos v2*, important for long-running optimizations, as well as optimizing the size of intermediate save-points.

[19] https://viztracer.readthedocs.io/en/latest/
[20] https://github.com/pympler/pympler

6. Graphical User Interface

As previously discussed, *Optimos v2* in its core can be used via a *CLI* or as a Python module. Both ways require at least intermediate technical expertise, which limits accessibility for non-technical users. However, business stakeholders and *process owners* are often non-technical. To make *Optimos v2* more accessible, a *Graphical User Interface* (*GUI*) was created. It provides the same functionality available to *CLI* users, internally invoking the same interface, but improves usability by offering a simplified, browser-based interface. The *GUI* was inspired by the design and functionality of the *pix-portal*[21], but is a standalone, self-contained application.

6.1 Architecture

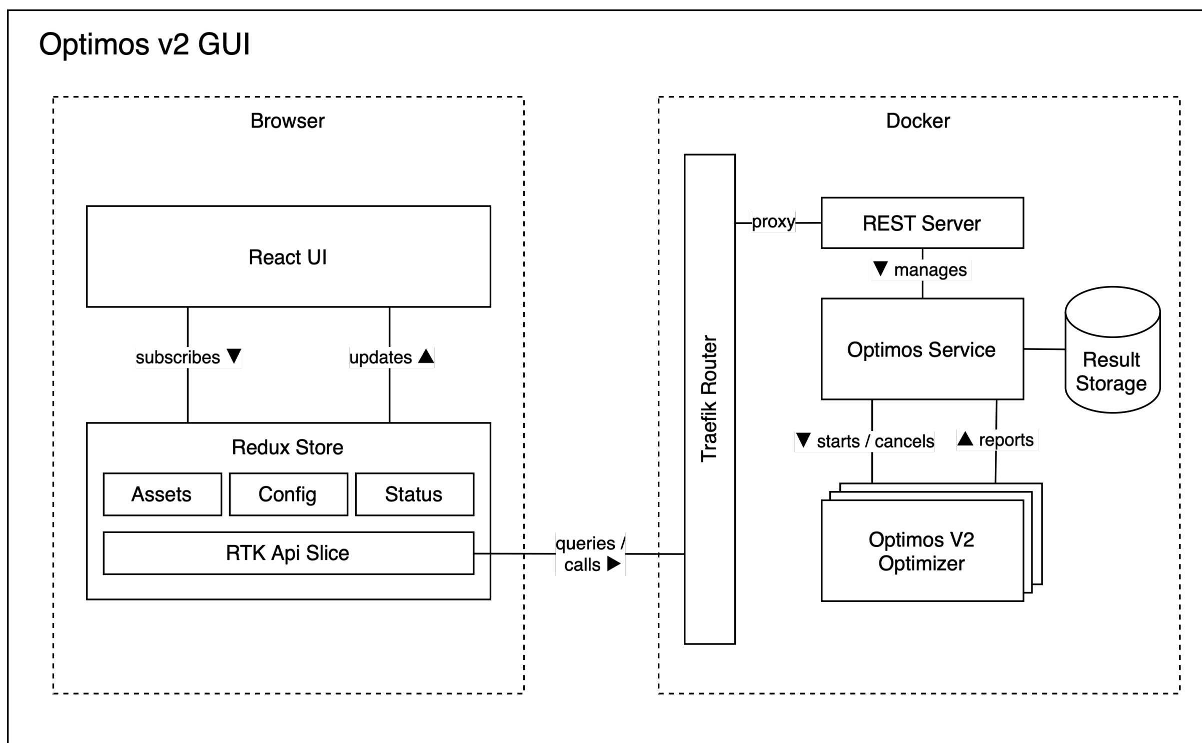


Figure 6. High-level overview of the *Optimos v2 GUI* architecture.

The *GUI* web app itself was created using *TypeScript*[22], a type-safe *JavaScript* superset, and the frameworks *React*[23], and *Redux*[24], with graphical elements from the *Mantine*[25] library.

[21] https://github.com/AutomatedProcessImprovement/pix-portal
[22] https://www.typescriptlang.org
[23] https://react.dev
[24] https://redux.js.org
[25] https://mantine.dev

It uses *Hypertext Transfer Protocol* (*HTTP*) calls to communicate with a *Python*-based *HTTP* Server, which follows the *Representational State Transfer* (*REST*) architectural style implemented in *fastapi*[26] framework. A *Traefik*[27] reverse proxy is used to provide *Transport Layer Security* (*TLS*) encryption for the *HTTP* calls (also known as *Hypertext Transfer Protocol Secure* (*HTTPS*)).

The *REST* Server communicates with the *Optimos v2 Service* which manages the running optimizations and allows creating or canceling optimization runs. It also stores the simulation results on the file system, so they can later be retrieved by the frontend.

All the above components (hosting of the web app, the *Traefik* proxy and the *Optimos v2* server components) are distributed and managed as *Docker*[28] containers, allowing fast deployment across diverse server environments.

Figure 6 shows an overview of this structure. Notably, the *GUI* also follows the *CQRS* architecture through the use of a *Redux Store*, which encapsulates an immutable state, containing the user's assets, the current optimization configuration, as well as the state of running simulations. To keep the latter updated in real-time, we use the *RTK Query*[29] library.

6.2 Usage

To interact with *Optimos v2*, users simply open the *Optimos v2* web page, i.e. the hosted variant at https://optimos.jannisrosenbaum.de. No login or additional setup is required.

The basis of each optimization is a set of assets: a simulation model with a *BPMN*, representing the process control flow, and a *XML* model, a JSON file with the simulation parameters, as described in section 2.2. The simulation models supported by *Optimos v2* can be generated by Simod following the instructions from the code repository[30] and uploaded via the `Upload asset` component. After selecting the desired simulation model in the *`Optimos v2`* `GUI`, users can define constraints to restrict the optimization process. As shown in fig. 7, the constraints are grouped into `Global`, `Scenario`, and `Resource Constraints`.

[26] https://fastapi.tiangolo.com
[27] https://traefik.io/traefik/
[28] https://www.docker.com
[29] https://redux-toolkit.js.org/rtk-query/overview
[30] https://github.com/AutomatedProcessImprovement/Simod

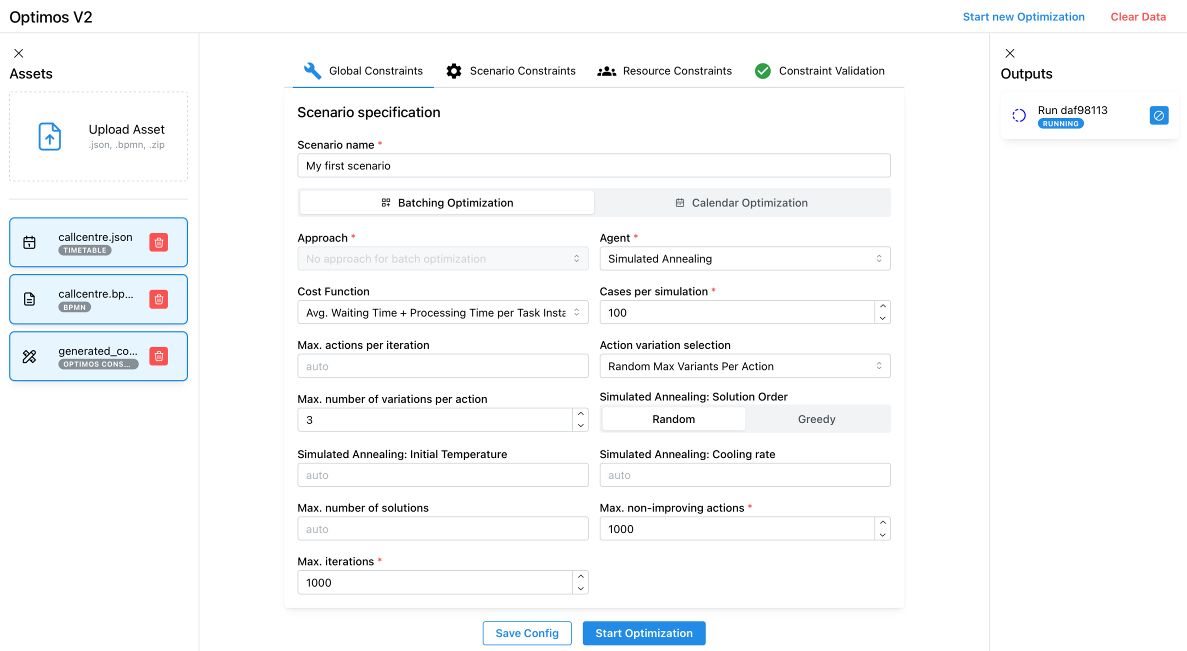


Figure 7. View of the *Optimos v2 GUI* setup page

The `Global Constraints` (the tab shown in fig. 7) relate to general optimization settings. They include the maximum number of iterations, the choice of the search algorithm (”agent”), and fine-tune many parameters, e.g., the initial *SA* temperature. Additionally, the cost function can be selected, i.e. whether fixed cost are considered, how many interventions are evaluated in parallel per iteration and how heuristics are selected.

Under the `Scenario Constraints` tab, users can set constraints on the batching applied. This includes maximum batch size, which tasks allow batching and define the discount function. The `Resource Constraints` ab is relevant only if the *Optimos v2 GUI* is used for resource calendar optimization, based on a reimplementation of the work of Berx [41]. This functionality is beyond the scope of this thesis.

Finally, the `Constraint Validation` tab displays and supports correction of all the potential errors in the constraint definitions, e.g., when a batching policy violates the defined maximum batch size. If all the constraints are valid (indicated in green), users can start the optimization by clicking the corresponding button.

Users can view the processing queue for both ongoing and completed simulations on the right side of the `Optimos v2 GUI` (see fig. 8). Clicking an entry opens a detailed optimization report. These reports show metrics such as simulated median cost, median time, and waiting times achieved at each iteration, helping users compare improvements to the original process.

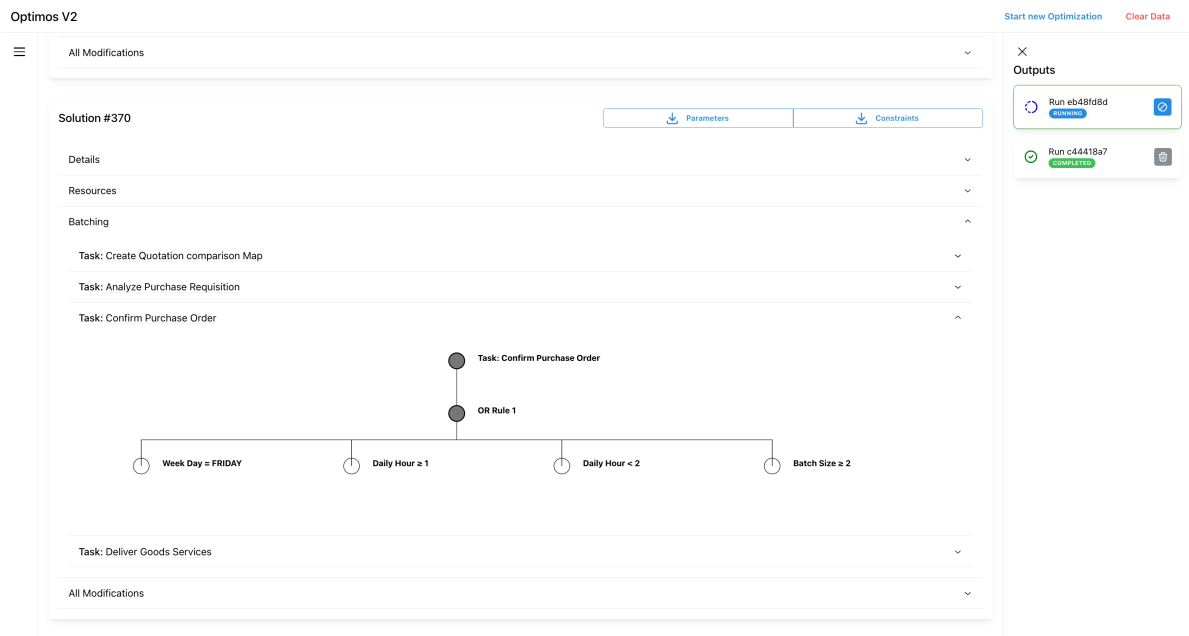


Figure 8. View of the *Optimos v2 GUI* results page

The reports also include detailed information for each resource and task. Visual aids such as a solution scatter plot display how costs and cycle times improve across iterations. A *tree view* gives a human-readable representation of the current batching policies (see fig. 8). This helps analysts understand how each iteration modifies the batching policies to enhance performance. Users may review these results and choose a batching policy from the solutions produced at each iteration, based on their preferences for balancing time and cost. *Optimos v2* GUI allows users to stop the optimization process at any point if they are satisfied with the results. Solutions can be exported as a JSON *simulation model* at any time, which in turn can be used as a new input for the *Optimos v2* GUI.

7. Conclusion

In this thesis, we proposed a heuristic-guided approach to discover and optimize batching policies, defining 19 domain-specific heuristics to balance the trade-offs between waiting time, processing effort, and cost. We integrated these heuristics into meta-heuristic search strategies, enhancing their ability to automatically refine batching constraints and discover a set of Pareto-optimal policies. The software was implemented using a sound architectural design and was statically verified and tested.

The evaluation, conducted on 10 real-life processes, demonstrated that heuristic-driven methods improve convergence by finding solutions closer to the reference Pareto front, increase diversity by providing a broader set of policies, and reduce or preserve the average process case cycle times by balancing batching trade-offs.

Furthermore, a *GUI* was implemented and deployed, making the full *Optimos v2* software accessible to non-technical users, and enables any process owner to optimize the batching policies in their *Business Processes*.

An avenue for future work is batching optimization in which batch activation is driven by data-aware conditions, adapting dynamically to process attributes rather than relying on fixed thresholds. For example, activation could depend on order type, urgency, or production constraints, allowing batches to have a specific size or trigger earlier based on real-time process conditions. Another promising direction is extending the heuristic and meta-heuristic approaches to support prioritization and multitasking policies, enabling batches to contain instances of different activities while ensuring optimal scheduling and resource allocation by dynamically adjusting execution priorities based on workload conditions.

References

[1] Dumas M., La Rosa M., Mendling J., and Reijers H. A. Fundamentals of Business Process Management. Berlin, Heidelberg: Springer, 2018. DOI: 10.1007/978-3-662-56509-4. (03/28/2025).

[2] Pufahl L. and Weske M. Batch Activity: Enhancing Business Process Modeling and Enactment with Batch Processing. *Computing* 101.12 (Dec. 2019), pp. 1909–1933. DOI: 10.1007/s00607-019-00717-4. (03/28/2025).

[3] López-Pintado O., Dumas M., and Berx J. Discovery, Simulation, and Optimization of Business Processes with Differentiated Resources. *Information Systems* 120 (Feb. 2024), p. 102289. DOI: 10.1016/j.is.2023.102289. (03/28/2025).

[4] Object Management Group. Business Process Model and Notation (BPMN) Specification, Version 2.0.2. 2013.

[5] Lashkevich K., Milani F., Chapela-Campa D., and Dumas M. Data-Driven Analysis of Batch Processing Inefficiencies in Business Processes. *Research Challenges in Information Science*. Ed. by Guizzardi R., Ralyté J., and Franch X. Cham: Springer International Publishing, 2022, pp. 231–247. DOI: 10.1007/978-3-031-05760-1_14.

[6] IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams. *IEEE Std 1849-2023 (Revision of IEEE Std 1849-2016)* (Sept. 2023), pp. 1–55. DOI: 10.1109/IEEESTD.2023.10267858. (05/10/2025).

[7] Potts C. N. and Kovalyov M. Y. Scheduling with Batching: A Review. *European Journal of Operational Research* 120.2 (Jan. 2000), pp. 228–249. DOI: 10.1016/S0377-2217(99)00153-8. (05/04/2025).

[8] Camargo M., Dumas M., and González-Rojas O. Learning Accurate Business Process Simulation Models from Event Logs via Automated Process Discovery and Deep Learning. *Advanced Information Systems Engineering*. Ed. by Franch X., Poels G., Gailly F., and Snoeck M. Cham: Springer International Publishing, 2022, pp. 55–71. DOI: 10.1007/978-3-031-07472-1_4.

[9] López-Pintado O., Dumas M., Yerokhin M., and Maggi F. M. Silhouetting the Cost-Time Front: Multi-objective Resource Optimization in Business Processes. *Business Process Management Forum*. Ed. by Polyvyanyy A., Wynn M. T., Van Looy A., and Reichert M. Cham: Springer International Publishing, 2021, pp. 92–108. DOI: 10.1007/978-3-030-85440-9_6.

[10] Boussaïd I., Lepagnot J., and Siarry P. A Survey on Optimization Metaheuristics. *Information Sciences*. Prediction, Control and Diagnosis Using Advanced Neural Computations 237 (July 2013), pp. 82–117. DOI: 10.1016/j.ins.2013.02.041. (03/28/2025).

[11] Weise T. Global Optimization Algorithms-Theory and Application. *Self-Published Thomas Weise* (2009).

[12] Kirkpatrick S., Gelatt C. D., and Vecchi M. P. Optimization by Simulated Annealing. *Science* 220.4598 (May 1983), pp. 671–680. DOI: 10.1126/science.220.4598.671. (05/10/2025).

[13] Schulman J., Wolski F., Dhariwal P., Radford A., and Klimov O. Proximal Policy Optimization Algorithms. Aug. 2017. DOI: 10.48550/arXiv.1707.06347. arXiv: 1707.06347 [cs]. (05/10/2025).

[14] Huang S. and Ontañón S. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings* 35 (May 2022). Comment: Accepted into the proceedings of International FLAIRS Conference Proceedings, Vol. 35 (2022). DOI: 10.32473/flairs.v35i.130584. arXiv: 2006.14171 [cs]. (05/02/2025).

[15] Michail P. and Christos K. Object Relational Mapping Vs. Event-Sourcing: Systematic Review. *Electronic Government and the Information Systems Perspective*. Ed. by Kő A., Francesconi E., Kotsis G., Tjoa A. M., and Khalil I. Cham: Springer International Publishing, 2022, pp. 18–31. DOI: 10.1007/978-3-031-12673-4_2.

[16] Fowler M. Event Sourcing. https://martinfowler.com/eaaDev/EventSourcing.html. Dec. 2005. (05/01/2025).

[17] Dahan U. Clarified CQRS. https://udidahan.com/2009/12/09/clarified-cqrs/. Dec. 2009. (05/10/2025).

[18] Young G. CQRS and Event Sourcing. https://web.archive.org/web/20101223082150/http://codebetter.com and-event-sourcing/. Dec. 2010. (05/10/2025).

[19] Gamma E., Helm R., Johnson R., and Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley, Oct. 1994.

[20] Kabbedijk J., Jansen S., and Brinkkemper S. A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software. *Proceedings of the 17th European Conference on Pattern Languages of Programs*. EuroPLoP ’12. New York, NY, USA: Association for Computing Machinery, July 2012, pp. 1–10. DOI: 10.1145/2602928.2603078. (05/01/2025).

[21] Fowler M. Patterns of Enterprise Application Architecture. Nineteenth printing. The Addison-Wesley Signature Series. USA: Addison-Wesley Longman Publishing Co., Inc., Nov. 2002.

[22] Liu J. and Hu J. Dynamic Batch Processing in Workflows: Model and Implementation. *Future Generation Computer Systems* 23.3 (Mar. 2007), pp. 338–347. DOI: 10.1016/j.future.2006.06.003. (03/28/2025).

[23] Pufahl L., Meyer A., and Weske M. Batch Regions: Process Instance Synchronization Based on Data. *2014 IEEE 18th International Enterprise Distributed Object Computing Conference*. Sept. 2014, pp. 150–159. DOI: 10.1109/EDOC.2014.29. (03/28/2025).

[24] Natschläger C., Bögl A., Geist V., and Biró M. Optimizing Resource Utilization by Combining Activities Across Process Instances. *Systems, Software and Services Process Improvement*. Ed. by O'Connor R. V., Umay Akkaya M., Kemaneci K., Yilmaz M., Poth A., and Messnarz R. Cham: Springer International Publishing, 2015, pp. 155–167. DOI: 10.1007/978-3-319-24647-5_13.

[25] Pufahl L. and Karastoyanova D. Enhancing Business Process Flexibility by Flexible Batch Processing. *On the Move to Meaningful Internet Systems. OTM 2018 Conferences*. Ed. by Panetto H., Debruyne C., Proper H. A., Ardagna C. A., Roman D., and Meersman R. Cham: Springer International Publishing, 2018, pp. 426–444. DOI: 10.1007/978-3-030-02610-3_24.

[26] Pflug J. and Rinderle-Ma S. Application of Dynamic Instance Queuing to Activity Sequences in Cooperative Business Process Scenarios. *International Journal of Cooperative Information Systems* 25.01 (Mar. 2016), p. 1650002. DOI: 10.1142/S0218843016500027. (03/28/2025).

[27] Pufahl L., Bazhenova E., and Weske M. Evaluating the Performance of a Batch Activity in Process Models. *Business Process Management Workshops*. Ed. by Fournier F. and Mendling J. Cham: Springer International Publishing, 2015, pp. 277–290. DOI: 10.1007/978-3-319-15895-2_24.

[28] Wen Y., Chen Z., Chen T., Liu J., and Kang G. A Particle Swarm Optimization Algorithm for Batch Processing Workflow Scheduling. *2012 Second International Conference on Cloud and Green Computing*. Nov. 2012, pp. 645–649. DOI: 10.1109/CGC.2012.67. (03/28/2025).

[29] Rabta B. and Reiner G. Batch Sizes Optimisation by Means of Queueing Network Decomposition and Genetic Algorithm. *International Journal of Production Research* 50.10 (May 2012), pp. 2720–2731. doi: 10.1080/00207543.2011.588618. (03/28/2025).

[30] Castillo F. and Gazmuri P. Genetic Algorithms for Batch Sizing and Production Scheduling. *The International Journal of Advanced Manufacturing Technology* 77.1 (Mar. 2015), pp. 261–280. doi: 10.1007/s00170-014-6456-5. (03/28/2025).

[31] Akhtar M. U., Raza M. H., and Shafiq M. Role of Batch Size in Scheduling Optimization of Flexible Manufacturing System Using Genetic Algorithm. *Journal of Industrial Engineering International* 15.1 (Mar. 2019), pp. 135–146. doi: 10.1007/s40092-018-0278-2. (03/28/2025).

[32] López-Pintado O., Rosenbaum J., Berx J., and Dumas M. Optimos: A Tool for Simulation-Driven Business Process Optimization. *Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Forum at BPM 2024*. Vol. 3758. CEUR Workshop Proceedings. Skovde, Sweden: CEUR-WS.org, Sept. 2024, pp. 126–130.

[33] Martin N., Solti A., Mendling J., Depaire B., and Caris A. Mining Batch Activation Rules from Event Logs. *IEEE Transactions on Services Computing* 14.6 (Nov. 2021), pp. 1908–1919. doi: 10.1109/TSC.2019.2912163. (03/28/2025).

[34] López-Pintado O., Halenok I., and Dumas M. Prosimos: Discovering and Simulating Business Processes with Differentiated Resources. *Enterprise Design, Operations, and Computing. EDOC 2022 Workshops*. Ed. by Sales T. P., Proper H. A., Guizzardi G., Montali M., Maggi F. M., and Fonseca C. M. Cham: Springer International Publishing, 2023, pp. 346–352. doi: 10.1007/978-3-031-26886-1_23.

[35] Beckmann N., Kriegel H.-P., Schneider R., and Seeger B. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’90. New York, NY, USA: Association for Computing Machinery, May 1990, pp. 322–331. doi: 10.1145/93597.98741. (05/01/2025).

[36] Audet C., Bigeon J., Cartier D., Le Digabel S., and Salomon L. Performance Indicators in Multiobjective Optimization. *European Journal of Operational Research* 292.2 (July 2021), pp. 397–422. doi: 10.1016/j.ejor.2020.11.016. (03/28/2025).

[37] Custódio A. L., Madeira J. F. A., Vaz A. I. F., and Vicente L. N. Direct Multisearch for Multiobjective Optimization. *SIAM Journal on Optimization* 21.3 (July 2011), pp. 1109–1140. doi: 10.1137/10079731X. (03/28/2025).

[38] van Rossum G., Warsaw B., and Coghlan A. PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/. July 2001. (05/11/2025).

[39] van Rossum G., Lehtosalo J., and Langa Ł. PEP 484 – Type Hints. https://peps.python.org/pep-0484/. Sept. 2014. (05/11/2025).

[40] Gorelick M. and Ozsvald I. High Performance Python: Practical Performant Programming for Humans. 3rd edition. O'Reilly Media, Apr. 2025.

[41] Berx J. Optimisation of Business Processes with Differentiated Resources. Master's thesis. Tartu Ülikool, 2023. (04/17/2025).

Appendices

List of Figures

1 Example BPMN model of a blood testing process in a hospital setting. 8
2 Basic BPMN symbols: (a) Activity, (b) XOR Gateway (Split/Join), and (c) Event. 9
3 High-level overview of the *Optimos v2* architecture. 35
4 Example of Pareto fronts and non-optimal solutions (All algorithms, *ACC - Parallel*) 42
5 Screenshot of the *TensorBoard* showing real-time optimization metrics. 44
6 High-level overview of the *Optimos v2 GUI* architecture. 46
7 View of the *Optimos v2 GUI* setup page 48
8 View of the *Optimos v2 GUI* results page 49

List of Tables

1 Characteristics of the real-life business processes. 39
2 **EQ1 Convergence** – Averaged Hausdorff Distance Results. 41
3 **EQ1 Diversity** – Purity Results. 41
4 **EQ2 Gain** – Cycle Time Improvement Results. 42
5 Project size by component (lines of code, comments, blanks). 43

List of Algorithms

1 Unified Hill Climbing and Simulated Annealing Optimization 32
2 Reinforcement Learning-Based Optimization 33

Acronyms

BP *Business Process*

BPA *Batch Processing Areas*

BPM *Business Process Model*

BPMN *Business Process Model and Notation*

BPMS *Business Process Management System*

BPO *Business Process Optimization*

BPS *Business Process Simulation*

CI *Continuous Integration*

CLI *Command Line Interface*

Co *Courier*

CQRS *Command Query Responsibility Segregation*

CSV *Comma-Separated Values*

CT *Cycle Time*

Doc *Doctor*

ES *Event Sourcing*

GUI *Graphical User Interface*

HC *Hill-Climbing*

HTTP *Hypertext Transfer Protocol*

HTTPS *Hypertext Transfer Protocol Secure*

IT *Idle Time*

KPI *Key Performance Indicator*

LOC *Lines of Code*

LT *Lab Technician*

PPO *Proximal Policy Optimization*

PT *Processing Time*

REST *Representational State Transfer*

RL *Reinforcement Learning*

SA *Simulated-Annealing*

TLS *Transport Layer Security*

WT *Waiting Time*

XES *eXtensible Event Stream*

XML *Extensible Markup Language*

XOR *Exclusive OR*

License

Non-exclusive license to reproduce the thesis and make the thesis public

I, Jannis Rosenbaum,

1. grant the University of Tartu a free permit (non-exclusive license) to

reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis

Optimization of Activity Batching Policies in Business Processes,

supervised by Orlenys López-Pintado, PhD & Marlon Dumas, PhD;

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons license CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Jannis Rosenbaum

15/05/2025