DECISIONBRAIN: AGENTIC SOLVER ORCHESTRATION FOR END-TO-END OPERATIONS RESEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Solving an operations research problem end to end from its natural-language statement is challenging: the requirements are implicit in the statement, solver performance varies across problem classes, and an optimal solution to an incorrect model may violate the requirements. Although language models and agent frameworks have facilitated such workflows, most existing systems lack an explicit representation for a solution strategy designed by the agent, restrict execution to specific solvers, and require the solving stage to verify its own output. We present DecisionBrain, an end-to-end agentic workflow that orchestrates heterogeneous solvers through strategy-level design over a validated method library. It returns a decision only after an independent feasibility review, which routes repairable failures back to the responsible stage. DecisionBrain and every open-source baseline run on the same DeepSeek-V4-Flash backbone. On the stress test suite Hard32-Fea, DecisionBrain returns a feasible decision on 81.3% of instances against 59.4% for the strongest baseline. On FrontierOR65-Fea, where a solver based on one monolithic mathematical optimization model already suffices, DecisionBrain reaches 86.2% against 84.6% for the strongest baseline. Ablations confirm that each mechanism contributes, and removing the algorithm library has the largest effect, 46.9 points on Hard32-Fea.

1 INTRODUCTION

Operations research problems arise wherever limited resources must be allocated under interacting constraints, and classical operations research supplies mature models, solvers, and domain-specific algorithms for them (Fourer et al., 1990; Bussieck & Meeraus, 2004; Hart et al., 2011; Dunning et al., 2017; Gurobi Optimization, LLC, 2024; Wouda et al., 2024). These methods operate on a mathematical model, and obtaining such a model from a natural-language statement requires identifying the decision variables, stating every requirement as a constraint, choosing the objective, and fixing the meaning and the units of the data (Astorga et al., 2025; Kong et al., 2026). The model must then be matched to a solver, and no single solver is effective across problems. Mixed-integer programming, constraint programming, and dedicated routing solvers are effective on different structures (Gurobi Optimization, LLC, 2024; Perron & Didier, 2025; Wouda et al., 2024). Even a returned decision must still be checked against the original statement, since a gap may remain between that statement and the model built from it.

Large language models and agent frameworks connect language understanding with code generation and tool use in one workflow (Yao et al., 2023; Schick et al., 2023; Gao et al., 2023; Wu et al., 2024). In operations research a language model can be trained or prompted to produce the model itself, turning a statement into an algebraic formulation that a solver receives (Huang et al., 2025a; Jiang et al., 2025). A language model can also search for a solution directly by proposing candidates and revising them using returned objective values (Yang et al., 2024; Liu et al., 2024). Multi-agent systems take a task from its statement and data to a decision, dividing the work among roles that interpret the statement, write the formulation, generate the solver code, and repair what the solver rejects (AhmadiTeshnizi et al., 2024; Xiao et al., 2024; Zhang et al., 2025), and some of them dispatch the resulting model among several solver backends (Li et al., 2026; Thind et al., 2025). Evaluation has shifted accordingly, from modeling accuracy on natural-language statements to benchmarks that execute systems and score the decisions they return end to end (Ramamonjison et al., 2022; Huang et al., 2025b; Kong et al., 2026).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We identify three limitations in most existing optimization-agent systems. First, they must build a monolithic mathematical optimization model of the task and translate it into executable code, which leaves strategy selection largely implicit. The language model is used as a translator rather than as a planner. Second, agents constrained to a monolithic model invoke only the solvers that accept its formalism, which leaves mature domain-specific methods inaccessible and the solution process unable to adapt to the instance, although algorithm selection and configuration have been studied for decades (Rice, 1976; Xu et al., 2008; Hutter et al., 2009; 2011; Kotthoff, 2014). In addition, the solver code these systems generate can contain invalid or hallucinated API calls (Chen et al., 2025), and the method used may remain unclear. Third, output correctness is not independently established. In many systems, the same stage that constructs a solution also verifies whether it is correct (Huang et al., 2024; Valmeekam et al., 2023). As a result, a program that runs to completion or reports an objective value is taken as evidence that the intended problem was solved, and a modeling failure may instead be treated as an implementation failure, which can direct repair to the wrong stage. The three limitations are linked, since without a monolithic model the agent must invoke methods correctly, and a correct invocation still does not guarantee a correct decision.

Our contributions address these limitations and validate the proposed DecisionBrain:

- **Strategy-level orchestration over a validated algorithm capability layer.** A design stage selects and composes heterogeneous solvers and other methods, each applied to the full problem, a subproblem, a relaxation, or a repair model. Each component of the design invokes a validated domain-specific method through the interface registered for that method, and the executed method is recorded. Also, the design stage admits a monolithic mathematical optimization model when one suffices.
- **Independent checking and responsibility-directed repair.** DecisionBrain writes its own feasibility checker per instance and executes it only in a review stage operationally separated from the stages that construct the candidate. If the decision does not pass the checker, review routes repairable failures to ALGORITHM DESIGN or SOLVING, while suspected infeasibility enters the confirmation procedure.
- **Empirical validation of the proposed workflow.** We introduce benchmarks derived from public data and published papers, and conduct evaluation on them. Hard32-Fea is a stress test of solver orchestration, and FrontierOR65-Fea contains tasks where a single solver already suffices. DecisionBrain reaches 81.3% constraint feasibility on Hard32-Fea against 59.4% for the strongest baseline and 86.2% on FrontierOR65-Fea against 84.6%. Nested ablations show that each of the three mechanisms contributes.

Without a monolithic mathematical optimization model, correctness can be assessed only from the decision returned, so the evaluation focuses on constraint feasibility. All code and data are open source.

2 PROBLEM DEFINITION

We consider optimization tasks specified by natural-language requirements and structured instance data (Kong et al., 2026). A task is represented as

$$\mathcal{I} = (R, D, B),$$

where R states the business objective, constraints, data semantics, and required output in natural language; D is the structured data of the instance; and B is the execution budget. The budget B comprises the end-to-end task budget B_{task} and the aggregate execution limit B_{solve} for the program launched in one SOLVING attempt.

The pair (R, D) implicitly defines an optimization problem

$$\mathcal{P}_{\mathcal{I}} = (\mathcal{X}_{\mathcal{I}}, \mathcal{F}_{\mathcal{I}}, \mathbf{f}_{\mathcal{I}}, \preceq_{\mathcal{I}}),$$

where $\mathcal{X}_{\mathcal{I}}$ is the decision space, $\mathcal{F}_{\mathcal{I}} \subseteq \mathcal{X}_{\mathcal{I}}$ is the feasible set defined by all requirements, $\mathbf{f}_{\mathcal{I}} : \mathcal{X}_{\mathcal{I}} \to \mathcal{Y}$ collects the task objectives, and $\preceq_{\mathcal{I}}$ is a preference relation on $\mathcal{Y}$, stated in R, by which objective vectors are compared. Writing $\prec_{\mathcal{I}}$ for its strict part, the set of optimal outcomes is

$$\mathcal{X}^{\star}_{\mathcal{I}} = \{x \in \mathcal{F}_{\mathcal{I}} \mid \mathbf{f}_{\mathcal{I}}(y) \not\prec_{\mathcal{I}} \mathbf{f}_{\mathcal{I}}(x) \ \ \forall\, y \in \mathcal{F}_{\mathcal{I}}\}.$$

R specifies both the objectives and how outcomes are compared, and the workflow makes no prior assumption about either. R may state one or several objectives and rank outcomes by any criterion it expresses. The form of $\mathcal{F}_\mathcal{I}$ is equally open, covering constraints that are not linear, continuous, differentiable, or written in any single modeling formalism.

A workflow reaches $\mathcal{P}_\mathcal{I}$ through models. A model is a tuple of the same form, written in a formalism that some solver accepts, together with a decoding map back into the decision space of the task. Its objective map has type $\mathbf{f}_M : \mathcal{X}_M \to \mathcal{Y}$, and $\preceq_M$ is a preference relation on $\mathcal{Y}$.

$$M = (\mathcal{X}_M, \mathcal{F}_M, \mathbf{f}_M, \preceq_M), \qquad \varphi_M \colon \mathcal{X}_M \to \mathcal{X}_\mathcal{I}.$$

M is complete when it represents the task in full,

$$\varphi_M(\mathcal{F}_M) = \mathcal{F}_\mathcal{I}, \qquad \mathbf{f}_M = \mathbf{f}_\mathcal{I} \circ \varphi_M, \qquad \preceq_M = \preceq_\mathcal{I}.$$

Three difficulties follow. First, M is not given with the task. A workflow must construct it from (R, D), and a misread requirement gives $\varphi_M(\mathcal{F}_M) \neq \mathcal{F}_\mathcal{I}$, a model of a different problem. Second, a complete M still has to be solved within B_{solve} during a SOLVING attempt, and no single method in the available set $\mathcal{A}$ is assumed to be best across all tasks and budgets. Third, writing $\mathcal{X}^\star_M$ for the optimal outcomes of M under $\preceq_M$, solving M certifies nothing about $\mathcal{P}_\mathcal{I}$ unless M is complete, since $x \in \mathcal{X}^\star_M \not\Rightarrow \varphi_M(x) \in \mathcal{F}_\mathcal{I}$.

DecisionBrain responds to each of the three. It replaces the single complete M by a design, a finite sequence of components with their scopes and budget shares,

$$\Pi = (a_i, \sigma_i, b_i)_{i=1}^{k}, \qquad \sum_{i=1}^{k} b_i \leq B_{\text{solve}},$$

where σ_i names the part of $\mathcal{P}_\mathcal{I}$ that a_i acts on. Any model used by a component may cover only its assigned scope, and a component may return a partial decision. The components run in order and their outputs compose into a candidate $x \in \mathcal{X}_\mathcal{I}$. Each method is drawn from a library that pairs it with a validated interface, $\mathcal{L} = \{(a, \iota_a) \mid a \in \mathcal{A}\}$ with $a_i \in \mathcal{A}$ and $(a_i, \iota_{a_i}) \in \mathcal{L}$, where ι_a fixes how the task data reaches a and how its output is read back. A candidate is returned only when a checker generated from R and executed apart from the components that produced it accepts, $c \colon \mathcal{X}_\mathcal{I} \to \{0, 1\}$ with x returned if and only if $c(x) = 1$. The only other returned outcome is a validated infeasibility conclusion.

3 DECISIONBRAIN

DecisionBrain organizes an optimization task as a sequence of six stages connected by structured artifacts, built around strategy-level orchestration over a validated algorithm library and independent verification. The workflow runs from INTAKE through PROBLEM CONTRACT, ALGORITHM DESIGN, SOLVING, and FEASIBILITY REVIEW to EXPLANATION. Every transition is gated by the Runtime Harness, the execution layer that launches the solver and checker programs and lets a stage advance only after its output matches that stage's contract. A rollback sends a rejected candidate back to the earlier stage responsible for the failure (Section 3.4).

3.1 MAIN WORKFLOW

Figure 1 contrasts an agent that uses one monolithic mathematical optimization model with DecisionBrain's design over components validated by the method library, and its independent review. Figure 3 gives the full execution flow, including its gates and rollback paths. Appendix A gives the input and output of each stage.

The stage sequence. INTAKE checks whether the task is specified well enough to model and asks for clarification where it is not. PROBLEM CONTRACT formalizes the decision variables, objective, constraints, and representations, and writes an executable feasibility checker. ALGORITHM DESIGN measures the instance and specifies the methods to run, each method's share of the per-attempt solving budget, and a fallback. SOLVING implements the design, runs the solvers, and returns a terminal outcome with its evidence. FEASIBILITY REVIEW checks structural conformance, then feasibility against the checker withheld from the constructing stages. EXPLANATION reports only the outcomes FEASIBILITY REVIEW has accepted.

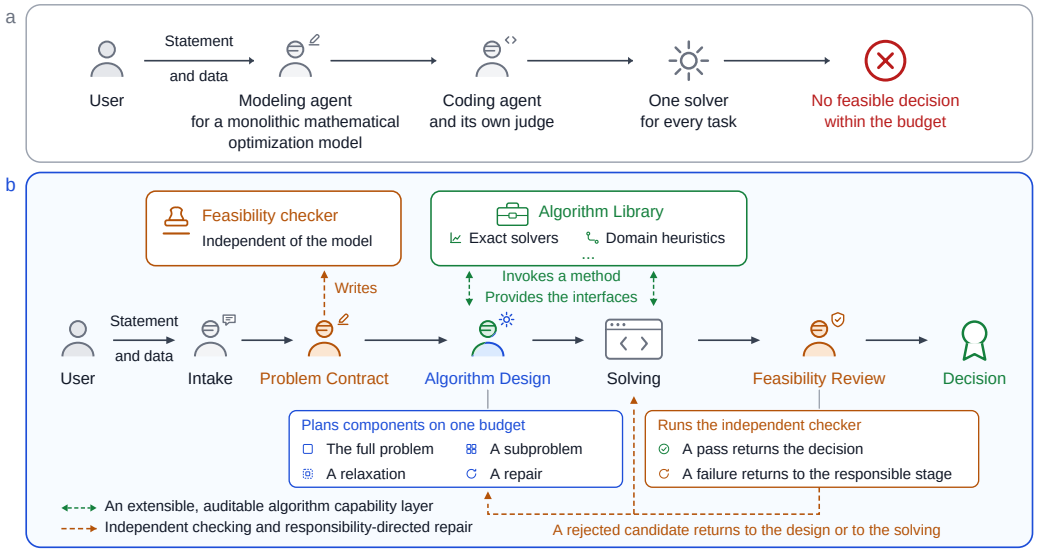


Figure 1: (a) An agent builds one monolithic mathematical optimization model from the statement and hands it to one solver. (b) DecisionBrain replaces that route with a design over components validated by a method library, and a review that runs a checker written by PROBLEM CONTRACT.

Gates and information partitioning. A stage advances only after the Runtime Harness validates its output against that stage's contract, so a malformed artifact returns to the same stage. For SOLVING and FEASIBILITY REVIEW the gate also checks that the artifact came from the process the harness launched. Appendix A lists the check each gate applies. In addition to validation, the contracts restrict what each stage receives to the artifacts they list.

3.2 STRATEGY-LEVEL ALGORITHM DESIGN

Current agents reduce modeling to the translation of the task into one monolithic model and its solver code. We replace this modeling stage with ALGORITHM DESIGN, which chooses the methods that apply, decides how they combine, and splits the per-attempt solving budget among them. A design is expressed in components, which belong to no single modeling formalism. The system checks each design against an explicit contract without constraining the form the design takes. A design can therefore contain several models, each restricted to part of the task. For example, a design may decompose a routing task into per-cluster routing, per-route sequencing, and residual assignment, and give each of these models its own time limit (Appendix G.1).

Component definition. A component is the unit in which ALGORITHM DESIGN specifies work, and four groups of attributes define it. *Method identity and guarantee:* the method family and which of three answer classes it delivers, a proven optimum, an incumbent from an unfinished exact method, or an incumbent from a search that proves nothing. *Scope:* the full task, a subproblem, a relaxation, or a repair model. *Capability source:* a concrete solver of a concrete package in the library of Section 3.3, or an implementation written from scratch by SOLVING. The recorded source bounds the guarantee a component may declare, since a hand-written implementation cannot inherit the optimality guarantee of a package solver. *Interface and resource specification:* how the component passes task data to its solver and reads the result back, and which budget share, stopping condition, and requirements it is assigned. A design of one component remains valid when a single method handles the whole task. Appendix B explains why several are often needed and how components may span packages.

Composition and accountability. A design of several components states the order in which they run, what each passes on, how the fixed per-attempt solving budget is split, the terminal outcomes permitted for each component, and when a partial result is accepted. The design must name the

component responsible for final feasibility, since a result can satisfy every component and still be infeasible for the task when each component covers only part of the constraints. Each design also carries a fallback, specified in the same component terms as the primary path. Figure 2 shows a single-component and a three-component design on one instance.

For a design of several components, the method each component names must be invoked correctly, which requires a capability layer that ties every component to a validated implementation.

3.3 Structured Algorithm Library

The algorithm library is the capability layer that ties each component to a validated implementation, and each method in it is reached through its own interface. Validation covers package versions, interfaces, availability, and result semantics. The library contains validated package manifests and solver-level entries. An entry records package identity and version, supported problem families, method class, interface, stopping controls, result fields, and the boundary of any guarantee it exposes.

Guide retrieval. Algorithm Design reads detailed documentation only for the packages and solvers it has decided to use. The design stage shortlists from a compact catalog, then reads a package guide for each shortlisted package and a solver guide for each solver it intends to call. Solving retrieves the same guide again to check the installed version and record the interface it executed. Interfaces are retrieved rather than reconstructed from memory. Since the guides for all candidate methods would exhaust the context window, reading full guides only for shortlisted packages keeps this cost tied to the packages Algorithm Design actually chooses. Appendix C gives the retrieval rules in full, and Appendix H the registered packages.

Traceable method use. Each component reports the method and guarantee of the solver it names and does not claim guarantees beyond those recorded. Solving records the package, solver, and version actually executed for each component, together with what the design declared. An advertised method can therefore be checked against the run (Appendix C).

3.4 Independent Review and Responsibility-Based Repair

DecisionBrain delegates acceptance decisions to an independent review stage. The independent Feasibility Review decides acceptance by executing a checker, and independent review means that the construction stages cannot access or execute the checker. The review then routes each repairable rejection to one repair target under a predefined attribution policy.

A checker generated for each task. During Problem Contract, DecisionBrain turns the formalized constraints into an executable feasibility checker. Each run therefore builds its acceptance test from its own contract. Only Feasibility Review executes the checker, and the stages that build the candidate have no access to it. The Runtime Harness launches the checker and records its output directly. This internal review checker is distinct from the hidden benchmark checker used only for external evaluation.

Checking logic. The review first checks structural conformance against the generated solution schema and then mathematical feasibility by executing the checker, so a candidate that parses but violates a constraint is rejected rather than accepted. It also classifies solver termination into four categories: a candidate is available, none was produced, a monolithic model was reported infeasible, or the harness deadline stopped the process. Each category carries different evidence forward (Appendix D).

Rollback and responsibility attribution. For a repairable rejection, the review assigns the failure to exactly one repair target according to a predefined attribution policy. Defects in the modeling decision, the choice of method, the component scope, or the budget go back to Algorithm Design. Solving handles defects in data transformation, implementation, or serialization. The message sent back to that stage contains the observable failure and the changes the repair must make, excluding the checker implementation. A rollback keeps the previous design, implementation, and outcome as a repair baseline, and the revised candidate is checked again under the same contract. A model level

infeasibility report is accepted as a validated infeasibility conclusion only after three consecutive reviews reach it, and each review runs without access to previous verdicts. Appendix D gives the handoff contents, the repair ledger, and the escalation rules. Figure 2 records one rollback and the attribution that selected its repair target.

3.5 The Method on One Instance

Figure 2 traces one complete run through the workflow of this section. The task is one instance of the Hard32-Fea suite of Section 4.2, a pickup-and-delivery problem with a cap of four vehicles.

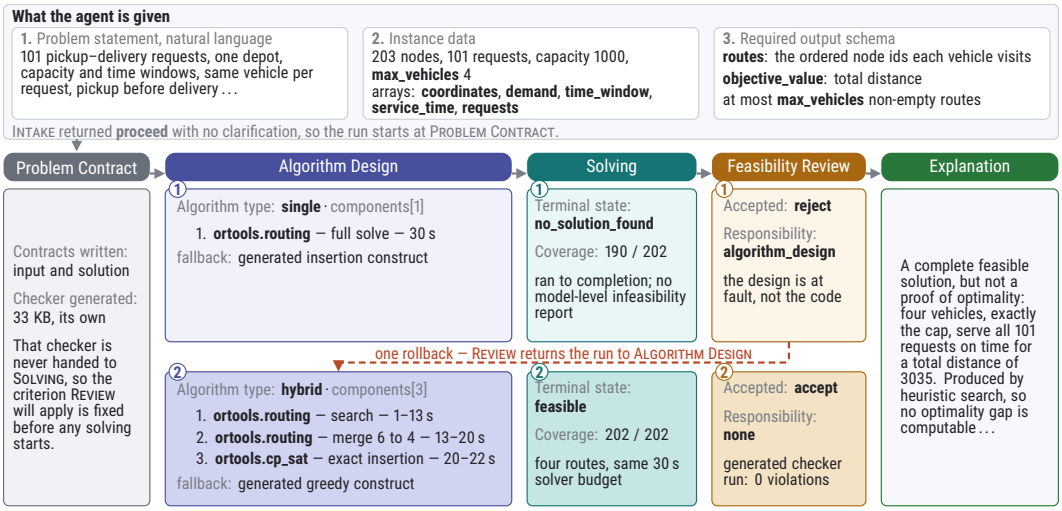


Figure 2: Workflow trace of one Hard32-Fea instance. Columns are the executed stages; the three stages that can repeat carry both passes. Every value is copied from the run's own artifacts.

The first design is a single component, OR-Tools routing (Perron & Furnon, 2025) on the whole problem. It ends without a candidate, and Feasibility Review attributes the failure to Algorithm Design rather than to Solving, since the execution record shows that the run used the OR-Tools routing solver the design named (Figure 7). The second design splits the same 30-second budget for that Solving attempt across three components. OR-Tools routing first searches for routes that cover every request, and a second routing pass merges the resulting six routes into four. CP-SAT then inserts the few remaining requests exactly, and a greedy construction serves as the fallback. The second design returns a candidate that passes the checker. Explanation reports it as feasible but not proven optimal, since it comes from heuristic search.

4 Experiments

We evaluate DecisionBrain on Hard32-Fea, FrontierOR65-Fea and its infeasible variant suite, all derived from public datasets. Each agent pipeline receives a task budget of 7,200 seconds.

4.1 Protocol and Baselines

All local runs use the same hardware, and DecisionBrain and the three open-source baselines use the same DeepSeek-V4-Flash (DeepSeek-AI, 2026) backbone. The closed-source commercial baseline is accessed through its web interface and does not expose its backbone or execution hardware. The Runtime Harness applies one aggregate execution limit to each solving attempt, shared by every component and solver call made by the submitted program: 600 seconds on FrontierOR65-Fea and its infeasible variant suite FrontierOR10-Inf, and 30 seconds on Hard32-Fea. The latter matches the regime with a fixed per-attempt solving budget used to construct the Hard32-Fea stress test. The tighter limit is intended to make differences in model design and solver choice more consequential within that regime. All evaluations start at Problem Contract with a deterministic Intake surrogate, and the Intake stage is not evaluated.

The acceptance criterion is constraint feasibility. A returned decision counts as feasible when it passes every constraint test of the hidden checker, whether or not the recomputed objective agrees with the reported one. Feasibility is graded by the hidden checker provided with each instance, which tests the returned decision independently of how the task was modeled and stays outside the agent workspace.

We compare DecisionBrain with four agent systems, the closed-source commercial Famou Agent Decision Platform (Baidu, 2026) and the open-source frameworks OR-LLM-Agent (Zhang et al., 2025), OptiMUS (AhmadiTeshnizi et al., 2024), and COOPA (Li et al., 2026). For baselines that do not accept the task format or emit the required output contract, fixed conversion stages map the task into their input and their candidates into that contract. Appendix F gives the hardware, the per-baseline adaptations, and the scoring conventions in full.

4.2 BENCHMARKS

FrontierOR65-Fea is a 65-instance suite selected from FrontierOR (Kong et al., 2026), whose tasks are drawn from published operations research papers, each with the model of its source paper and a feasible reference. Under the reported protocol and solver limit, a single general-purpose solver already solves most of these instances. We also include a separate 10-instance synthetic infeasibility set, and use it to measure whether a system recognizes an empty feasible set.

Hard32-Fea is a targeted stress test of method orchestration under a fixed solver budget, constructed from three public instance families—Taillard job-shop (Taillard, 1993), the Gehring–Homberger vehicle routing problem with time windows (VRPTW) (Homberger & Gehring, 2005), and the Li–Lim pickup and delivery problem with time windows (PDPTW) (Li & Lim, 2003). Each instance carries one added limit. The particular monolithic model handed to Gurobi misses it within the solver budget, and the method established for that family meets the limit. For most instances, the reference method used in selection is not available in the DecisionBrain library. Appendix E gives the construction rules, the selection records, and the task distributions of all three evaluation suites.

4.3 RESULTS ON FRONTIEROR65-FEA

DecisionBrain returns a feasible decision on 56 of the 65 instances, or 86.2%, ahead of OR-LLM-Agent at 84.6%, COOPA at 73.8%, Famou at 66.2%, and OptiMUS at 32.3% (Table 1). DecisionBrain and OR-LLM-Agent are within 1.6 points of each other, and including or excluding checker timeouts changes which of the two ranks first. On this suite, DecisionBrain is comparable to the systems that pass one monolithic model to a particular general-purpose solver. Among feasible instances where both systems have valid objective gaps, DecisionBrain finds the better objective more often than each baseline. Solution quality and runtime are reported in Appendix F. COOPA dispatches one monolithic model among four optimizer agents, each built on a different package, yet it trails OR-LLM-Agent, which passes its model to one solver. We conjecture that exploring additional alternatives adds overhead on instances already handled by one solver. DecisionBrain may face the same overhead because it chooses among far more methods.

On the 10-instance synthetic infeasibility set, DecisionBrain explicitly identifies 5 instances as infeasible, against COOPA's 4, Famou's 3, and none from OptiMUS and OR-LLM-Agent. Counting explicit conclusions or no-candidate outcomes, DecisionBrain has 7, ranking second. OptiMUS avoids an incorrect feasible return on all 10 only by returning no candidate anywhere.

Table 1: Constraint feasibility on FrontierOR65-Fea. The best value in each row is in bold.

Metric	DecisionBrain	COOPA	Famou	OR-LLM-Agent	OptiMUS
Constraint-feasibility rate	**86.2%**	73.8%	66.2%	84.6%	32.3%
Rate excluding checker timeouts	86.2%	76.2%	67.2%	**87.3%**	33.3%
Rate among completed candidates	**94.9%**	76.2%	86.0%	93.2%	91.3%
Constraint-feasible candidates	**56**	48	43	55	21
Candidates violating a constraint	3	14	7	4	**0**
No-candidate instances	6	**0**	14	4	40

4.4 RESULTS ON HARD32-FEA

DecisionBrain returns a feasible decision on 26 of the 32 instances, or 81.3%, against 59.4% for Famou, 15.6% for COOPA, and none for OptiMUS and OR-LLM-Agent (Table 2). Every candidate DecisionBrain returns passes the independent check, while COOPA, Famou, and OptiMUS each return some candidates that violate a constraint. OptiMUS and OR-LLM-Agent, which formulate each instance as one monolithic linear program (LP) or mixed integer linear program (MILP) and pass the model to Gurobi without the review loop, return no feasible decision on any instance. Restricted to the same Gurobi, DecisionBrain still solves 11, showing that Hard32-Fea does not reduce to a comparison between Gurobi and non-Gurobi methods. By family, DecisionBrain solves all 6 job-shop instances against Famou's 5 and 15 of 20 pickup-and-delivery instances against 14. On VRPTW it solves 5 of 6 against Famou's 0, missing only the 800-customer instance, where the task budget runs out. The accepted decisions come from CP-SAT, PyVRP, OR-Tools routing, ALNS, and a dispatching rule, and the composition varies by instance (Appendix G.1).

Table 2: Constraint feasibility on Hard32-Fea. DecisionBrain (Gurobi) is the restricted arm; — denotes no returned candidate.

Metric	DecisionBrain	DecisionBrain (Gurobi)	COOPA	Famou	OptiMUS	OR-LLM-Agent
Constraint-feasibility rate	**81.3%**	34.4%	15.6%	59.4%	0.0%	0.0%
Rate among completed candidates	**100.0%**	**100.0%**	26.3%	82.6%	0.0%	—
Job-shop deadline ($n = 6$)	**6**	3	4	5	0	0
VRPTW min-fleet ($n = 6$)	**5**	0	0	0	0	0
PDPTW fleet-cap ($n = 20$)	**15**	8	1	14	0	0
Constraint-feasible candidates	**26**	11	5	19	0	0
Candidates violating a constraint	**0**	**0**	14	3	2	**0**
No-candidate instances	**6**	21	13	9	30	32

4.5 ABLATIONS

The baseline comparison tests the three mechanisms together. The nested ablations examine whether strategy-level design changes solving behavior, whether a wider algorithm library improves feasibility and reduces repairs, and whether the review stage prevents infeasible candidates from being returned.

All ablations run on Hard32-Fea, where the fixed solver budget makes differences among the arms observable. The design and review ablations take the full system restricted to Gurobi as their reference arm (Table 3). In the design comparison, both arms have the same Gurobi solver pool and differ only in whether strategy-level design is replaced by a modeler that writes one monolithic Gurobi model. The restricted arm is the fairer reference for the monolithic paradigm, since a system built on one monolithic model cannot compose methods and can choose only among solvers that accept its formalism, gaining little from a larger pool. The restricted arm also keeps the large effect of the algorithm library from hiding the effect of the review. The library ablation compares the restricted pool against the full pool.

The first ablation replaces the design stage with a modeler that builds one monolithic mathematical optimization model in Gurobi and keeps everything else unchanged. The clearest difference lies in the process: over all 32 instances, the monolithic model arm makes 1,983 stage errors against 784, about 2.5 times as many. The design arm has fewer errors on 31 of the 32 instances. On the instances each arm solves, the monolithic model arm also takes longer on average, 4,455.4 seconds against 3,621.6. The feasibility rate changes less, falling by 9.4 points from 34.4% to 25.0%. Both arms run under the same 7,200-second task limit, and most runs in either arm end at that limit without a decision. Within that limit the modeler can keep retrying, which may keep the feasibility gap small at the cost of the extra errors and time. The next ablation widens the pool beyond Gurobi.

Narrowing the package pool to Gurobi removes the methods the design chooses among and has the largest effect. The feasibility rate falls by 46.9 points, from 26 instances to 11. The narrowed arm also repairs more, 56 responsibility-directed repairs against 11, on 25 instances against 6. Every

instance solved under the narrowed pool is also solved under the full pool. A wider pool does not change which candidates may be returned. The final ablation instead tests whether review changes which generated candidates are returned.

In this ablation, FEASIBILITY REVIEW leaves the aggregate feasibility rate unchanged, with 11 instances in both arms, although each arm solves 5 instances the other does not. It keeps infeasible candidates from being returned. Without it, the arm returns 16 candidates, 5 of which violate a constraint, so feasibility among returned candidates falls from 100.0% to 68.8%. The unreviewed arm is faster on the instances it solves, 2,646.9 seconds against 3,621.6, consistent with the absence of rollback and repair triggered by review. Full per-arm results are in Appendix F.4.

Table 3: Nested ablations on Hard32-Fea; Δ is the change relative to the parent arm.

Arm	Feasible instances	Feasibility rate	Δ (pp)	Feasible returns	Mean errors, feasible	Mean time, feasible (s)
Full system	26	81.3%	—	100.0%	10.5	1,982.1
− algorithm library	11	34.4%	−46.9	100.0%	16.6	3,621.6
− design stage	8	25.0%	−9.4	100.0%	44.0	4,455.4
− feasibility review	11	34.4%	0.0	68.8%	15.4	2,646.9

5 DISCUSSION AND LIMITATIONS

The ablations point to three distinct roles. Strategy-level design determines the solution process, and its programs build several restricted models instead of one model of the whole task (Table 27). The algorithm library determines which solvers the solution process can invoke, and the methods behind accepted decisions differ across families and instances (Table 28). FEASIBILITY REVIEW determines whether an output may be returned (Table 24). Strategy-level design and the algorithm library are mutually dependent, since a design reaches only the methods the library exposes, while the independent review decides whether their output may be returned. Removing any one of the three changes a different quantity, and therefore they are complementary. Only the algorithm library shows a family pattern. Without it the feasible count falls in every family, and no VRPTW instance is solved. No comparable family pattern appears in the main metrics for ALGORITHM DESIGN or FEASIBILITY REVIEW.

Limitations. First, the evidence is limited in scale. Hard32-Fea covers three families and 32 instances, and all open-source runs use DeepSeek-V4-Flash. We kept Hard32-Fea small given the cost of running five systems under a 7,200-second task limit, and the short window in which DeepSeek-V4-Flash was available. Second, the workflow costs extra time on instances such as those in FrontierOR65-Fea that are already handled well by a monolithic model. The pipeline still designs, checks, and repairs on these instances, even when those additional stages provide limited benefit.

6 CONCLUSION

An end-to-end agent can solve operations research tasks specified in natural language without being restricted to the solver based on one monolithic mathematical optimization model. On Hard32-Fea, DecisionBrain returns a feasible decision on 81.3% of instances against 59.4% for the strongest baseline, and it is comparable to systems that pass one monolithic model to a particular general-purpose solver on FrontierOR65-Fea, where such a solver already suffices for most instances. DecisionBrain orchestrates heterogeneous solvers by separating three roles that existing systems leave implicit or absent. Strategy-level design decides per instance how the task is solved, the algorithm library ties each chosen method to a validated interface, and an independent review decides whether the result may be returned.

Two directions follow. The first is to recognize per instance when the solver based on one monolithic model is enough, since exploring other methods adds run time on FrontierOR65-Fea, where such a solver already handles most instances well. The second is to reuse compositions that succeeded on earlier instances, since every design is currently built from scratch.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic data sets, provide feedback on research methodology and experiments, implement methods, assist with translation, clean and reformat datasets, support qualitative and thematic data analysis, and interpret results. We did not use generative AI tools to develop theoretical models or conceptual frameworks or to propose or refine hypotheses. Formulating mathematical claims, providing critical ingredients for proving mathematical claims, and assisting in the writing of proofs are not applicable to this work. We also used generative AI tools to create or edit software code, create or modify scientific figures, draft parts of the paper, edit the paper to improve readability, suggest its structure, and format references. We have reviewed all AI-assisted work. All AI-written code was reviewed and tested by the authors, every reported feasibility result was obtained from the independent hidden benchmark checkers rather than from AI output, and all reported numbers, text, and figures were checked by the authors against the raw run records. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects, user studies, or personal data. All instances come from public operations research benchmarks: FrontierOR65-Fea and FrontierOR10-Inf are derived from FrontierOR, released under the CC-BY-4.0 license and drawn from published papers, and Hard32-Fea is built from the public Taillard, Gehring–Homberger, and Li–Lim instance families by adding one constraint to each instance; none of these contains personally identifiable information. The commercial baseline was accessed through its standard user interface under a regular account. We see no direct potential for harm specific to this work beyond that of automated optimization tools in general, and we note that constraint-feasible decisions returned by such systems should still be reviewed by domain experts before deployment.

REPRODUCIBILITY STATEMENT

Section 3 and Appendices A–D specify every stage of DecisionBrain, including what enters each stage, how designs are composed, how the algorithm library is retrieved and recorded, and how review hands off and escalates; Appendix H lists the version, solvers, and interfaces of every package in the library. Section 4.1 and Appendix F give the hardware, task and solver budgets, backbone model, scoring conventions, and the adaptation of each baseline. The source FrontierOR benchmark is publicly available (Kong et al., 2026); Appendix E.1 describes the construction of FrontierOR65-Fea and FrontierOR10-Inf, and Appendix E.2 gives the construction rules, selection records, and per-instance parameters of Hard32-Fea, whose source instances are public. Appendix F.4 records the configuration of each ablation arm. The commercial baseline can be rerun through its public web interface, but its closed backend prevents exact reproduction of the reported runs from the released artifacts. Anonymized source code, the Hard32-Fea instances, and the run records are available at https://anonymous.4open.science/r/DecisionBrain-anonymous-6434.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 577–596. PMLR, 2024. URL https://proceedings.mlr.press/v235/ahmaditeshnizi24a.html.

Nicolás Astorga, Tennison Liu, Yuanzhang Xiao, and Mihaela van der Schaar. Autoformulation of mathematical optimization models using LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. URL https://arxiv.org/abs/2411.01679.

Baidu. Famou Agent Decision Platform. https://cloud.baidu.com/product/famou.html, 2026.

Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. Enabling research through the SCIP optimization suite 8.0. *ACM Transactions on Mathematical Software*, 49(2):1–21, 2023. doi: 10.1145/3585516.

Michael R. Bussieck and Alexander Meeraus. General algebraic modeling system (GAMS). In *Modeling Languages in Mathematical Optimization*, Applied Optimization, pp. 137–157. Springer, 2004. doi: 10.1007/978-1-4613-0215-5_8.

Yujia Chen, Mingyu Chen, Cuiyun Gao, Zhihan Jiang, Zhongqi Li, and Yuchi Ma. Towards mitigating API hallucination in code generated by LLMs with hierarchical dependency aware. In *Companion Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 468–479, 2025. doi: 10.1145/3696630.3728569.

Julien Coupey, Jean-Marc Nicod, and Christophe Varnier. VROOM v1.14, vehicle routing open-source optimization machine. http://vroom-project.org/, 2024.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Iain Dunning, Joey Huchette, and Miles Lubin. JuMP: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi: 10.1137/15M1020575.

Robert Fourer, David M. Gay, and Brian W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36(5):519–554, 1990. doi: 10.1287/mnsc.36.5.519.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10764–10799, 2023. URL https://proceedings.mlr.press/v202/gao23f.html.

Gurobi Optimization, LLC. Gurobi optimizer reference manual. https://www.gurobi.com, 2024.

William E. Hart, Jean-Paul Watson, and David L. Woodruff. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation*, 3(3):219–260, 2011. doi: 10.1007/s12532-011-0026-8.

Jörg Homberger and Hermann Gehring. A two-phase hybrid metaheuristic for the vehicle routing problem with time windows. *European Journal of Operational Research*, 162(1):220–238, 2005. doi: 10.1016/j.ejor.2004.01.027.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 73(6):2986–3009, 2025a. doi: 10.1287/opre.2024.1233.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/8b4add8b0aa8749d80a34ca5d941c355-Abstract-Conference.html.

Xuhan Huang, Qingning Shen, Yan Hu, Anningzhe Gao, and Benyou Wang. LLMs for mathematical modeling: Towards bridging the gap between natural and mathematical languages. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 2678–2710, 2025b. doi: 10.18653/v1/2025.findings-naacl.146.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009. doi: 10.1613/jair.2861.

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization (LION)*, pp. 507–523, 2011. doi: 10.1007/978-3-642-25566-3_40.

Caigao Jiang, Xiang Shu, Hong Qian, Xingyu Lu, Jun Zhou, Aimin Zhou, and Yang Yu. LLMOPT: Learning to define and solve general optimization problems from scratch. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=9OMvtboTJg.

Minwei Kong, Chonghe Jiang, Ao Qu, Wenbin Ouyang, Zhaoming Zeng, Xiaotong Guo, Zhekai Li, Junyi Li, Yi Fan, Xinshou Zheng, Xi Jing, Yikai Zhang, Zhiwei Liang, Seonghoo Kim, Runqing Yang, Zijian Zhou, Sirui Li, Han Zheng, Wangyang Ying, Ou Zheng, Chonghuan Wang, Jinglong Zhao, Hanzhang Qin, Cathy Wu, Paul Pu Liang, Jinhua Zhao, and Hai Wang. FrontierOR: Benchmarking LLMs' capacity for efficient algorithm design in large-scale optimization, 2026. URL https://arxiv.org/abs/2605.25246. Dataset repository: https://huggingface.co/datasets/SmartOR/FrontierOR.

Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *AI Magazine*, 35 (3):48–60, 2014. doi: 10.1609/aimag.v35i3.2460.

Chuanhao Li, Xiaoan Xu, Dirk Bergemann, Ethan X. Fang, Yehua Wei, and Zhuoran Yang. COOPA: A modular LLM agent architecture for operations research problems, 2026. URL https://arxiv.org/abs/2606.27611.

Haibing Li and Andrew Lim. A metaheuristic for the pickup and delivery problem with time windows. *International Journal on Artificial Intelligence Tools*, 12(2):173–186, 2003. doi: 10.1142/S0218213003001186.

Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. Large language models as evolutionary optimizers. In *IEEE Congress on Evolutionary Computation*, 2024. URL https://arxiv.org/abs/2310.19046.

Stephen Maher, Matthias Miltenberger, Jo ao Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. PySCIPOpt: Mathematical programming in Python with the SCIP optimization suite. In *Mathematical Software – ICMS 2016*, volume 9725 of *Lecture Notes in Computer Science*, pp. 301–307, 2016. doi: 10.1007/978-3-319-42432-3_37.

Laurent Perron and Frédéric Didier. CP-SAT, version 9.12. https://developers.google.com/optimization/cp/cp_solver/, 2025.

Laurent Perron and Vincent Furnon. OR-Tools, version 9.12. https://developers.google.com/optimization/, 2025.

Rindranirina Ramamonjison, Timothy Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong Zhang. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *Proceedings of the NeurIPS 2022 Competitions Track*, volume 220 of *Proceedings of Machine Learning Research*, pp. 189–203, 2022. URL https://proceedings.mlr.press/v220/ramamonjison23a.html.

John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976. doi: 10.1016/S0065-2458(08)60520-3.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.

Éric Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993. doi: 10.1016/0377-2217(93)90182-M.

Raghav Thind, Youran Sun, Ling Liang, and Haizhao Yang. OptimAI: Optimization from natural language using LLM-powered AI agents, 2025. URL https://arxiv.org/abs/2504.16918.

Karthik Valmeekam, Matthew Marquez, and Subbarao Kambhampati. Can large language models really improve by self-critiquing their own plans? In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023. URL https://openreview.net/forum?id=gGQfkyb0KL.

Niels A. Wouda and Leon Lan. ALNS: a Python implementation of the adaptive large neighbourhood search metaheuristic. *Journal of Open Source Software*, 8(81):5028, 2023. doi: 10.21105/joss.05028.

Niels A. Wouda, Leon Lan, and Wouter Kool. PyVRP: A high-performance VRP solver package. *INFORMS Journal on Computing*, 36(4):943–955, 2024. doi: 10.1287/ijoc.2023.0055.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Awadallah, Ryen W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. In *First Conference on Language Modeling (COLM)*, 2024. URL https://openreview.net/forum?id=tEAF9LBdgu.

Ziyang Xiao, Dongxiang Zhang, Yangjun Wu, Lilin Xu, Yuan Jessica Wang, Xiongwei Han, Xiaojin Fu, Tao Zhong, Jia Zeng, Mingli Song, and Gang Chen. Chain-of-Experts: When LLMs meet complex operations research problems. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/d45ee77826332c100a1e15f7765b99ff-Abstract-Conference.html.

Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. doi: 10.1613/jair.2490.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024. URL https://arxiv.org/abs/2309.03409.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=tvI4u1ylcqs.

Bowen Zhang, Pengcheng Luo, Genke Yang, Boon-Hee Soong, and Chau Yuen. OR-LLM-Agent: Automating modeling and solving of operations research optimization problems with reasoning LLM, 2025. URL https://arxiv.org/abs/2503.10009.

A STAGE REFERENCE

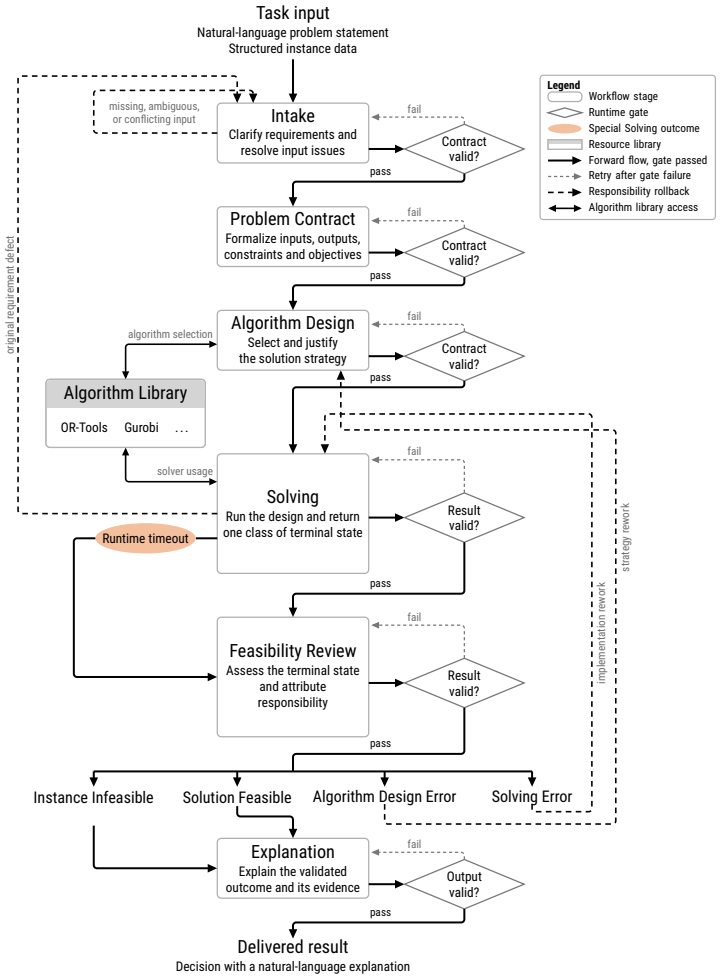


Figure 3: The full DecisionBrain workflow: forward execution, the gate each stage must pass, the algorithm library interaction, the four terminal states of SOLVING, and the responsibility-based rollbacks.

Section 3.1 states what each stage is responsible for. Table 4 adds the input and output of each stage and the check its gate applies. A stage whose output fails the gate receives a structured issue list and runs again, so the flow never advances past an unsatisfied gate. Only the process launched by the Runtime Harness can produce the result file of SOLVING or FEASIBILITY REVIEW. The harness records the digests of that process's files in a receipt (Figure 7), and the gate checks them again before the file is handed on. A stage therefore cannot check one candidate and submit another.

Table 4: Input, output, and gate check of each stage.

Stage	Input	Output	Gate check
INTAKE	Natural-language task statement and structured instance data	Clarified, complete problem definition that preserves the user's requirements	Output file exists, parses as strict JSON, and satisfies the stage contract

Stage	Input	Output	Gate check
PROBLEM CONTRACT	Original task and data; clarified definition	Task- and instance-specific contract and schemas (to construction and review); feasibility checker (to review only)	Same as INTAKE
ALGORITHM DESIGN	Statement, clarified definition, and source data; per-attempt solving budget (read-only) and algorithm catalog, injected by the harness	Algorithm-design contract: what to implement and why it fits	Same as INTAKE
SOLVING	Clarified problem, source data, input and solution contracts, algorithm design	Terminal outcome with evidence: a structurally valid candidate, a termination without candidate, or a model level infeasibility report	Contract check, and the solver result matches the sha256 digests of the harness receipt
FEASIBILITY REVIEW	Contract, design, terminal outcome, any candidate, execution evidence	Validated decision or validated infeasibility conclusion with evidence; otherwise a rollback (Section 3.4)	Contract check, and the checker result matches the sha256 digests of the harness receipt
EXPLANATION	Requirement context and accepted results	Final business-facing explanation to the user	Output file exists and parses as strict JSON, with no field-level contract

B DESIGN COMPOSITION

Section 3.2 defines components and how they compose. Figure 4 shows the design contract of the second pass of the worked-example run, with its candidates, components, fallback, and read-only budget schedule.

C ALGORITHM LIBRARY RETRIEVAL AND RECORDING

Section 3.3 states the retrieval rules and the execution record. This appendix shows them on the worked-example run of Figure 2. Figure 5 gives what that run retrieved at each level, and Figure 6 the catalog it received. Figure 7 gives a per-component execution record from the first pass and, separately, a harness receipt from the second pass. The receipt illustrates how the Runtime Harness binds a solver file, its input, and its solution to the process it launched.

D REVIEW HANDOFFS AND ESCALATION

Section 3.4 describes FEASIBILITY REVIEW, attribution, and escalation. This appendix gives the handoff and escalation rules and the structures FEASIBILITY REVIEW exchanges. Table 5 lists the evidence each terminal outcome of SOLVING carries to FEASIBILITY REVIEW. A model-level infeasibility report concerns a particular model and is not yet a conclusion about the task. FEASIBILITY REVIEW promotes it to a validated infeasibility conclusion only when the model covers every constraint in the problem contract and the escalation rule below is satisfied. Some failures keep SOLVING from producing a reviewable outcome at all: a program that cannot run, an unusable solver call, or output that fails the stage's own contract. These stay inside that stage and never enter attribution. After repeated failure to prove infeasibility, a revised design must include a time-limited proving attempt on a model covering every constraint wherever scale and structure permit. A different attribution between two infeasibility verdicts resets the count. Figure 8 shows the handoff issued by the first review of the worked-example run. It also shows the append-only ledger entry the handoff is compressed once a later review has judged the repair. That entry holds identifiers, a solve summary, and the outcome, without prose or checker evidence.

```
{ "problem_family": "routing",
  "diagnosis": { "decision_type": "route",
    "key_difficulty": "101 paired requests into <= 4 routes; neighbourhood
                       search stalls on the same missing request" },
  "scale": { "primary_items": 101, "resources": 4,
             "estimated_variables": 45600, "risk": "high",
             "confidence": "medium",
             "reason": "recomputed from data/instance.json" },
  "candidates": [
    { "solver": "ortools/routing", "method_class": "metaheuristic",
      "recommendation": "primary" },
    { "solver": "ortools/cp_sat", "method_class": "exact", ... "backup" },
    { "source": "generated", "method": "greedy", ...     "backup" } ],
  "selection": { "kind": "hybrid",
    "components": [
      { "component_id": "c1_search4_fullcoverage",
        "role": "main search, coverage first",
        "package": { "id": "ortools", "version": "9.12.4544" },
        "solver_id": "routing", "optimality": "heuristic_or_incumbent",
        "capability_mapping": [ 4 x { business_requirement,
                                      manifest_capability, evidence } ],
        "integration": { primary_api, input_mapping, result_mapping,
                         runtime_control, random_seed } },
      { "component_id": "c2_merge_repair_fullcoverage",
        "role": "merge a complete 6-route solution down to 4 routes",
        "solver_id": "routing", ... },
      { "component_id": "c4_exact_conditional",
        "role": "absorb <= 2 missing requests into one route exactly",
        "solver_id": "cp_sat", "optimality": "exact_or_incumbent", ... } ] },
  "strategy": { "representation": ..., "objective": ..., ...,
    "stopping": "30 s read-only: c1 1-13 | c2 13-20 | c4 20-22 |
                 fallback 22.5-26.5 | output 26.5-28.5 | margin >= 1.5" },
  "fallback": {
    "trigger": "no complete candidate after c1, c2, c4 and t <= 22.5 s",
    "components": [ { "component_id": "fb_gen_full_construct",
                      "source": "generated", "method": "greedy" } ] } }
```


Figure 4: Algorithm-design contract of the second pass of the worked-example run (abridged).

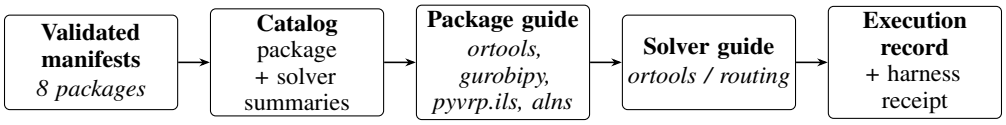


Figure 5: Retrieval hierarchy of the algorithm library. Italic lines give what the first design pass of the worked-example run (Figure 2) retrieved at each level.


```
{ "count": 8,
  "algorithms": [
    { "id": "alns", ... }, { "id": "gurobipy", ... },
    { "id": "job_shop_lib", ... },
    { "id": "ortools",
      "status": "experimental", "kind": "python_library",
      "summary": "OR-Tools 9.12.4544 package integration covering
                  CP-SAT, linear solving, min-cost flow, and ...",
      "problem_families": ["resource_scheduling", "routing", ...],
      "method": "optimization_toolkit", "method_class": "mixed",
      "solvers": [
        { "id": "cp_sat",
          "summary": "Solve integer constraint models with CP-SAT
                      and proof-aware feasible, optimal, or
                      infeasible statuses." },
        { "id": "linear_solver", ... }, { "id": "min_cost_flow", ... },
        { "id": "routing",
          "summary": "Build and search vehicle-routing models with
                      routing dimensions, local search, and time
                      limits." } ] },
    { "id": "pyjobshop", ... }, { "id": "pyscipopt", ... },
    { "id": "pyvrp.ils", ... }, { "id": "rsome", ... } ] }
```


Figure 6: Algorithm catalog injected into ALGORITHM DESIGN in the worked-example run (abridged).

execution record (SOLVING, pass 1)


```
{ "component_id":
      "ortools_routing_pdptw",
  "planned_package_id":  "ortools",
  "planned_solver_id":   "routing",
  "executed_package_id": "ortools",
  "executed_solver_id":  "routing",
  "method": "vehicle_routing_search",
  "version": "9.12.4544",
  "guide_consulted": true,
  "availability_check": { "ok": true },
  "seed": 0,
  "runtime_limit_seconds": 27.0,
  "fallback_used": false,
  "deviation_reason": "fallback not
      triggered: under 3.5 s left" }
```


harness receipt (SOLVING, pass 2)


```
{ "outcome_source":
      "runtime_solver_execution",
  "solver_file":     "solver.py",
  "solver_sha256":   "73dcf6d4...",
  "input_file":      "input.json",
  "input_sha256":    "fb702ec9...",
  "solution_file":   "solution.json",
  "solution_sha256": "d8a46f60...",
  "exit_code": 0,
  "elapsed_seconds": 25.12,
  "timed_out": false }
```


Figure 7: An execution record from pass 1 and, separately, a harness receipt from pass 2 of the worked-example run (abridged; hashes truncated). The checker run of FEASIBILITY REVIEW produces a receipt of the same form.

Table 5: Classification of SOLVING terminal outcomes.

Class	Solver status	Candidate	Evidence passed to review
Candidate available	Optimal, feasible, or an incomplete search with an incumbent	Present, structurally valid	Termination, objective information, and the candidate
No candidate	Time limit, unknown, no solution found, or infeasible-or-unbounded	Absent	Termination and diagnostics for why none was returned
Model-level infeasibility report	A monolithic model was reported infeasible	Absent	Solver termination, evidence that the model covers the complete problem contract, and any solver-supplied infeasibility evidence
Harness timeout	The outer deadline terminates the process	Possibly produced earlier	Timeout location, elapsed time, exit evidence, any preserved candidate

handoff (review, pass 1)

```
{ "decision": "reject",
  "responsibility": "algorithm_design",
  "remediation_handoff": {
    "requirement_ids": [
      "full_request_coverage",
      "request_pairing_precedence",
      "vehicle_capacity", "time_windows",
      "vehicle_fleet_limit", ... ],
    "change_ids": [
      "reassess_algorithm_family",
      "reassess_complete_formulation",
      "fleet_decomposition_strategy",
      "budget_rebalance_with_required_repair",
      "fallback_full_coverage_capability",
      "conditional_exact_attempt" ],
    "problem_requirement": "each request
        served once by one vehicle,
        pickup first; windows; capacity;
        <= 4 routes",
    "observed_solution_behavior": "ended
        normally at 26 s; best plan
        covers 190 of 202 nodes; no
        solution file",
    "responsibility_reason": "Solving ran
        the design faithfully; the design
        has no component that guarantees
        coverage or proving component",
    "required_changes": [ 5 x "..." ] } }
```

ledger entry (after pass 2)

```
{ "attempt": 1,
  "responsibility": "algorithm_design",
  "requirement_ids": [ 6 ids ],
  "change_ids": [ 6 ids ],
  "solver_summary": {
    "status": "no_solution_found",
    "diagnosis": "search ended at
        26.0 s; best 190/202, no
        complete candidate; not a
        proof",
    "methods": [
      "vehicle_routing_search" ] },
  "outcome": "accept",
  "next_responsibility": null }
```

Figure 8: Handoff issued by the first review of the worked-example run, and the ledger entry to which it is compressed once the second review has accepted the repair (abridged).

E BENCHMARK CONSTRUCTION

This section gives in full the construction summarized in Section 4.2: the data sources, the retained task distributions, the selection criterion, and the selection records of all three evaluation suites.

E.1 FRONTIEROR65-FEA AND FRONTIEROR10-INF

E.1.1 DATA SOURCE

FrontierOR (Kong et al., 2026) collects about 180 paper-level tasks, each tied to an optimization problem from a published operations research paper. Agent-visible inputs are a natural-language statement, structured instance data, and input and solution schemas. Evaluator-only materials are the mathematical model, reference code and solution, and a checker written for the structure of the source paper. Reference objectives are used only for relative quality comparison.

Dataset coverage. Tables 6 and 7 give the problem classes, model types, and reported model scale of the 65 retained tasks, using the benchmark's own labels. Reported scale describes each paper's reference formulation, not the instance files, and is strongly right-skewed (median far below mean). Its lower extremes reflect modeling style rather than easy problems. The task with a single constraint is a 10-million-variable knapsack, and the task with a single variable is a column-generation master. We therefore use scale neither to weight nor to stratify any result.

The table uses the following abbreviations: traveling salesperson problem (TSP), mixed integer programming (MIP), integer programming (IP), binary integer programming (BIP), distributionally robust optimization (DRO), quadratic programming (QP), mixed integer quadratic programming (MIQP), constraint programming (CP), nonlinear programming (NLP), mixed integer nonlinear programming (MINLP), and semidefinite programming (SDP).

Table 6: Retained task distribution by problem class and model type. The 27 tasks in the aggregated "Other" class are decomposed in Table 8.

Problem class	Count	Model type	Count
Graph Optimization	5	MIP	24
Scheduling (Machine / Crew / Workforce)	5	IP	12
Vehicle Routing & TSP	5	BIP	8
Packing / Cutting Stock	4	Stochastic/Robust/DRO	8
Lot Sizing / Inventory	3	QP/MIQP	5
Network Design & Flow	3	CP	3
Portfolio Optimization	3	NLP/MINLP	2
Vehicle Scheduling	3	Bilevel/Multi-level	2
Energy & Unit Commitment	2	SDP	1
Facility Location	2		
Stochastic / Robust / Bilevel Optimization	2		
Cutting Stock / Production Planning	1		
Other (multiple detailed classes)	27		
Total	65	Total	65

Table 7: Reported model scale of the 65 retained tasks. The figures are the per-task averages published in FrontierOR metadata for each paper's reference model, not counts extracted from the selected instance files.

Reported quantity	Minimum	Median	Mean	Maximum
Variables	1	69,536	2,429,599	100,000,000
Integer variables	0	30,383	478,685	10,000,000
Constraints	1	11,004	557,166	6,929,616

The aggregated "Other" class. The 27 retained tasks grouped under "Other" in Table 6 fall into the detailed classes of Table 8.

Table 8: Detailed breakdown of the aggregated "Other" problem class.

Detailed class	Count
Container Terminal	2
General MIP	2
Project Scheduling	2
Set Covering / Partitioning	2
Assembly Line Balancing	1
Assignment	1
Clustering	1
Cooperative Control	1
Generalized Assignment	1
Grouping / Partitioning	1
Knapsack	1
Network Flow	1
Nurse Rostering	1
Parallel Machine Scheduling	1
Quadratic Assignment	1
Quadratic Knapsack	1
Quadratic Optimization	1
Resource Assignment	1
Resource Loading	1
Revenue Management	1
Structural Design	1
Submodular Optimization	1
Transportation	1
Total	27

E.1.2 Benchmark Construction

Each task ships five large instances, and running all of them is not affordable, so the benchmark is reduced by an explicit rule to one instance per retained task.

Feasible-reference task set (FrontierOR65-Fea).

1. Take each task's instance with the largest serialized size.
2. Keep it only if the bundled reference solution passes the bundled checker and the verdict matches the recorded feasibility label. Otherwise fall back to the next largest instance, and drop the task if none passes.
3. Stratify the surviving tasks by problem class and model type and keep one task per stratum (the smaller median instance size, ties broken by paper identifier).

The rule yields 65 tasks. Step 2 requires a passing reference solution, so the set measures whether an agent produces and validates a feasible outcome, not how often instances are infeasible.

Synthetic infeasibility task set (FrontierOR10-Inf). Five structurally distinct source tasks each yield two variants (Table 9). The data-level variant keeps the statement and changes a parameter so that a necessary aggregate bound fails. The constraint-level variant keeps the data and adds a contradictory constraint to the visible contract. Each contradiction is short and independent of any generated implementation.

Table 9: Construction of the 10 synthetic infeasibility cases.

Source task	Data-level variant	Constraint-level variant
VRP with capacity and time windows (`desaulniers2010`)	Total demand exceeds fleet capacity	Total-delivery upper bound below required demand
Pickup and delivery with loads (`ropke2009`)	Positive pickup loads, zero-capacity vehicles	Zero vehicle load despite positive pickups
Nurse rostering (`rahimian2017`)	A nurse's minimum working time exceeds its maximum	Coverage of 51 nurses with 50 available
Facility assignment (`freling2003`)	All facility capacities zero, demand positive	One customer both assigned and not assigned to a facility
Unit commitment (`knueven2020`)	All generator maximum outputs zero, demand positive	All generators off while positive demand must be balanced

Visible-contract corrections. For 5 tasks the visible statement could not yield an accepted solution under any correct reading, a defect the reference-agreement filter cannot detect. We revised only the visible statement, to say what the checker already enforces. Checkers, reference materials, and data are untouched (Table 10).

Table 10: Visible-statement corrections.

Task	Statement before correction	What the checker requires
`morrison2016`	Ordinary vertex coloring	Set-covering model; every selected color class a maximal independent set
`xie2019`	Quay-crane profiles as free choices	Profiles generated from the input, filtered by a stated rule, indexed per vessel
`amaldi2013`	Hyperplane encoding unstated	The encoding the checker parses
`bollapragada2001`	Sign convention of equilibrium coefficients unstated	The sign convention the checker uses
`gruson2021`	Index base of solution keys unstated	The index base the checker uses

E.2 Hard32-Fea: Feasible Decision Generation under an Added Constraint

Section 4.2 introduces Hard32-Fea. This subsection gives its sources, limits, selection criterion, and per-instance record.

E.2.1 Data Source

The 32 instances reuse three public families unchanged: Taillard job-shop instances (Taillard, 1993), with the recorded best makespan upper bound ub; Gehring–Homberger RC2 VRPTW instances (Homberger & Gehring, 2005) from the SINTEF distribution, with the published best fleet count; and Li–Lim pickup-and-delivery instances (Li & Lim, 2003). The families put the constraint on three decision structures: a makespan deadline on a disjunctive schedule, and a route cap on fleet sizing without and with same-vehicle pickup-before-delivery pairing. Each is close to a common operational requirement: finishing production by a delivery date, serving customers with the smallest fleet, and carrying paired pickup-and-delivery requests, as in courier or on-demand transport services. The benchmark therefore has narrow coverage across problem classes but is deliberately challenging within them (Table 11). Objectives are never compared across instances.

Table 11: Coverage of the Hard32-Fea set: source instances and scale, the single constraint placed on each family, and the reference method anchoring its selection boundary. Here ub is the best makespan upper bound recorded with each Taillard instance, and K a vehicle count.

Task family	Count	Source instances and scale	Constraint at the boundary	Reference method
Job-shop deadline	6	Taillard ta41, ta51, ta61 at 30×20, 50×15, 50×20 jobs $\times$ machines, each at two deadlines	makespan $\leq D$, $D = \mathrm{round}(\theta \cdot \mathrm{ub})$, $\theta \in \{1.1, 1.2\}$	CP-SAT
VRPTW min-fleet	6	Gehring–Homberger RC2, 400, 600, and 800 customers	number of routes $\leq K$, K = published minimum fleet	PyVRP
PDPTW fleet-cap	20	Li–Lim large instances, 100–200 requests	number of routes $\leq K$, K from the reference free-fleet computation	VROOM
Total	32			

E.2.2 Benchmark Construction

The added limit. Job-shop instances receive a makespan deadline $D = \mathrm{round}(\theta \cdot \mathrm{ub})$ with $\theta \in \{1.1, 1.2\}$ on three base instances, and the objective is that same makespan. VRPTW instances receive a route cap K equal to the published best fleet count, which a bin-packing lower bound also attains, so the cap has no slack. Pickup-and-delivery instances receive a cap K equal to the default configuration's free-fleet route count, a measured rather than certified value. Every instance's limit is listed in Table 13.

Selection. Selection was a deterministic pre-run, neither an LLM run nor a DecisionBrain run, with a 30-second solver budget on each side. One side is a MILP handed to Gurobi, warm-started from a construction heuristic (Table 12). The other is the method established for that family in its own literature: CP-SAT (Perron & Didier, 2025) for job-shop, PyVRP (Wouda et al., 2024) for VRPTW, and VROOM (Coupey et al., 2024) for pickup-and-delivery. An instance is kept when the MILP does not meet its limit within the budget and that method does. Instances the construction heuristic alone meets are excluded, and the criterion was frozen before the runs. The criterion need not continue to hold under a different solver configuration. On the three job-shop instances at $\theta = 1.2$, the selection run found that the default Gurobi configuration did not reach feasibility within the budget. In the later Gurobi-only evaluation, one of these instances was solved under a different configuration; this does not change the construction criterion.

Table 12: The default solver configuration used in selection, by family. An instance is retained only if the configuration misses the limit within 30 seconds.

Task family	Gurobi MIP	Warm start
Job-shop deadline	Disjunctive model minimizing makespan, with pairwise big-M ordering of operations on each machine	Best priority-dispatch schedule among SPT, LPT, MWKR, LWKR, LMFT, and random rules
VRPTW min-fleet	Two-index arc-flow model with big-M time windows, minimizing the number of depot departures	Complete solution of a route-minimization construction (sequential insertion with an ejection pool)
PDPTW fleet-cap	Three-index vehicle–arc model with time-window arc pruning, arc-wise tightened big-M, vehicle symmetry breaking, `MIPFocus`$= 1$, `Heuristics`$= 0.3$	Partial solution of a pickup-and-delivery insertion heuristic run for 10 seconds, covering 76–89% of requests

Task structure. Each instance is repacked into the task structure used by FrontierOR65-Fea, so the protocol of Section 4.1 applies unchanged. Each instance adds the limit, written into the visible

statement as a business requirement. It also adds a hidden checker that decides every business constraint (completeness, the cap, capacity, time windows, depot return, and for pickup and delivery same-vehicle service with pickup first) in integer arithmetic with no tolerance, and a certified witness, the reference method's decision, which passes that checker on all 32 instances.

What the criterion fixes. Both sides of the criterion are external to DecisionBrain: a MILP under a default Gurobi configuration, and a method published for the problem family. They establish that a domain method reaches the limit within the solver budget where the default MILP configuration does not. They leave open what an agent delivers within the task budget (Section 4.4). On 12 instances the reference method is one the algorithm library also contains. On the other 20 it is VROOM, which the library does not contain.

Per-instance construction record. Table 13 gives, for every instance, the limit and the best value the default solver configuration reached for the same quantity during selection. Job-shop instance names carry the multiplier θ.

Table 13: Per-instance construction record for Hard32-Fea. The middle column is the added constraint's value; the right column is the best value the default solver configuration (Table 12) reached for that same quantity during selection, which exceeds the limit on every instance by construction. For pickup-and-delivery, where the MIP returned no solution, it is the value of the warm-start heuristic.

Instance	Limit	Default solver configuration
Job-shop deadline ($n = 6$): the makespan is bounded by the deadline		
ta41_th1.1	2206	2522
ta41_th1.2	2406	2528
ta51_th1.1	3036	3955
ta51_th1.2	3312	3910
ta61_th1.1	3155	4148
ta61_th1.2	3442	4148
VRPTW minimum fleet ($n = 6$): the number of vehicles used is bounded by the fleet cap		
RC2_4_6	8	9
RC2_6_6	11	13
RC2_6_7	11	12
RC2_8_3	15	17
RC2_8_6	15	16
RC2_8_7	15	16
PDPTW fleet cap ($n = 20$): the number of vehicles used is bounded by the fleet cap		
lc1_2_1	20	32
lc1_2_10	18	32
lc1_2_5	20	36
lc1_2_6	20	35
lc1_2_7	20	33
lc1_2_8	20	32
lc1_2_9	18	35
lc2_2_10	6	19
lc2_2_2	6	18
lc2_2_6	6	18
lc2_2_7	6	19
lc2_2_8	6	18
lc2_2_9	6	20
lr108	9	21
lr207	2	14
lr208	2	14
lrc1_2_10	13	28
lrc1_2_8	14	30

Table 13 (continued)

Instance	Limit	Default solver configuration
lrc2_2_8	4	21
lrc2_2_9	4	17

F Experimental Detail and Full Results

This section gives the hardware and protocol, the per-baseline adaptations, the secondary quality and runtime results, and the per-arm ablation tables behind Sections 4.3–4.5.

F.1 Experimental Setup

All runs used one workstation: an Intel Core i9-13900K (24 physical cores, 32 logical CPUs, one socket and NUMA node), 125 GiB of RAM, Ubuntu 24.04.3 LTS with Linux 7.0.0-28. Up to 8 tasks ran concurrently, each with a 3-core solver budget that the Runtime Harness divided among concurrent routes, at least 1 core each. Each solving attempt submits one program to the Runtime Harness. The aggregate execution limit for that program, shared by all components and solver calls within it, is 600 seconds on FrontierOR65-Fea and FrontierOR10-Inf and 30 seconds on Hard32-Fea (Section 4.1); the 7,200-second task budget covers all stages, solving attempts, and rollbacks.

Evaluation task initialization. Across all three evaluation suites every task starts at Problem Contract from a deterministic surrogate Intake artifact built only from the visible statement and instance data. It records objective, decisions, constraints, scale, and required output, and adds no checker, reference, model, method, or code. All systems thus receive the same task definition, and open-ended clarification is kept out of the solving evaluation.

Solving Evaluation. Stage failures and harness timeouts are recorded per case so the batch continues. No candidate, a rejected candidate, a checker failure, and a harness failure are recorded as distinct outcomes, and a model level infeasibility report is compared with reference feasibility, so a report on a feasible task earns no credit.

Two parts of the external check. Candidates that pass every constraint but report an objective the checker recomputes differently count as feasible (Section 4.1) and are excluded from gap statistics. Feasibility and solution quality are therefore evaluated separately, and the objective mismatches are reported in Section F.2.

For a constraint-feasible candidate whose reported objective agrees with the checker recomputation, let z denote that objective and $z_{\mathrm{ref}} \neq 0$ the reference objective. We compute the signed relative gap as

$$\mathrm{gap}(z, z_{\mathrm{ref}}) = \begin{cases} \dfrac{z - z_{\mathrm{ref}}}{|z_{\mathrm{ref}}|}, & \text{for minimization,} \\ \dfrac{z_{\mathrm{ref}} - z}{|z_{\mathrm{ref}}|}, & \text{for maximization.} \end{cases}$$

The objective direction is read from the visible statement. Lower is better, and a negative gap means that the candidate improves on the reference. Some references are best-known rather than optimal, so the gap is relative to the known reference.

F.1.1 Baseline

The four baselines and the output-conversion stage are introduced in Section 4.1. This subsection states how each was run on all three suites. Where required, the input- and output-conversion stages are fixed LLM calls using DeepSeek-V4-Flash. Each call has a 600-second limit outside the 7,200-second task budget, leaving the full task budget to the baseline's own pipeline. No output-conversion call failed. For every no-candidate or constraint-violating outcome, we inspected the raw baseline output and confirmed that the candidate was already absent or invalid before conversion.

DecisionBrain's repair loop has no round limit and stops only at the task limit, so where a baseline exposes an internal attempt counter we raised it until runs end on the shared time limit. Baselines without one ran as distributed. Famou admits a single round per task and has no such setting.

Famou Agent Decision Platform. Famou is operated through its web interface (https://cloud.baidu.com/product/famou.html), one question-and-answer round per task, with no trace, stage timing, or machine-readable status. Its times are measured outside the platform and include browser automation and scheduling overhead, and an infeasibility conclusion is counted only where the reply states it.

OR-LLM-Agent. OR-LLM-Agent is a single-thread pipeline that builds a model, generates Gurobi code, executes it, and repairs the program. Data are staged in the workspace and a fixed conversion step produces the required output.

OptiMUS. A manager coordinates modeling, programming, and evaluation for LP and MILP. Each task is converted to OptiMUS's input decomposition, with numeric data in a workspace file and the same conversion step on output. Its evaluator's diagnoses of the current model are not counted as conclusions about the instance. Two instances per suite fail in input conversion before timing starts. They remain in the feasibility denominator as no-candidate outcomes but are excluded from runtime statistics.

COOPA. COOPA generates candidate models, keeps the most confident one, and dispatches it to an optimizer agent with a specialized role. Data are staged in the workspace, the statement goes to the manager, and confidence-based iterative selection is enabled. Its manager iteration cap was raised so runs end on the task limit, and a fixed formatter produces the required output.

F.2 Results on FrontierOR65-Fea and FrontierOR10-Inf

F.2.1 Results on FrontierOR65-Fea

Table 1 and Section 4.3 give the feasibility results. This subsection adds what the table does not show.

There are six candidates that satisfy every constraint but report an objective the checker recomputes differently for DecisionBrain, four each for COOPA and Famou, three for OR-LLM-Agent, and one for OptiMUS.

OptiMUS's 91.3% among completed candidates is conditioned on 23 instances. It returns no candidate on 40 tasks for three reasons: an input conversion failed to produce a model, an internal solver error occurred, or a solve returned no incumbent. None of these failures is retried.

Solution quality. Table 14 pairs each baseline with DecisionBrain on the instances both solve feasibly with a reported gap. DecisionBrain's mean gap is smaller against OR-LLM-Agent and OptiMUS and larger against COOPA and Famou. The comparisons of means with COOPA and Famou are dominated by one instance (`taninmis2022`, gap 296.56); removing it reverses both comparisons. The instance-level counts put DecisionBrain ahead on all four pairings.

Runtime. Table 15 shares one denominator across systems. DecisionBrain is slowest. On instances solved by both, it is within seven percent of COOPA (2,269.1 against 2,133.1 seconds) but far slower than OR-LLM-Agent (2,447.9 against 834.7 on 51 instances), consistent with the overhead of composed designs and repeated review on a suite where a solver based on one monolithic model suffices. Famou's times include browser-automation overhead.

F.2.2 Results on FrontierOR10-Inf

Table 16 gives the full counts and runtimes behind the results of Section 4.3.

Table 14: Solution quality on instances that DecisionBrain and the other system both solve feasibly and for which both report a gap.

Metric	COOPA	Famou	OR-LLM-Agent	OptiMUS
Paired instances	36	29	43	19
DecisionBrain mean gap	8.1465	10.1138	**−0.0644**	**−0.0418**
Other system mean gap	**0.2183**	**4.4474**	3.2551	4.5530
DecisionBrain mean gap, excl. `taninmis2022`	**−0.0940**	**−0.1165**	—	—
Other system mean gap, excl. `taninmis2022`	0.2439	0.9383	—	—
DecisionBrain strictly better	**20**	**18**	**20**	**10**
Tied	14	6	16	6
Other system strictly better	2	5	7	3

Table 15: Runtime over FrontierOR65-Fea.

Metric	DecisionBrain	COOPA	Famou	OR-LLM-Agent	OptiMUS
Mean runtime (s)	2,958.0	2,430.6	**817.7**	921.4	1,311.0
Median runtime (s)	2,523.0	2,129.7	**440.9**	730.1	1,103.3
Mean, constraint-feasible instances only (s)	2,535.9	2,316.1	**495.2**	840.9	1,147.3

Table 16: Behaviour on the 10-instance synthetic infeasibility set. Runs that exhausted the task limit are counted at that limit of 7,200 seconds.

Metric	DecisionBrain	COOPA	Famou	OR-LLM-Agent	OptiMUS
Explicit infeasibility conclusions	**5 (50%)**	4 (40%)	3 (30%)	0 (0%)	0 (0%)
Explicit conclusion or no candidate	7 (70%)	6 (60%)	5 (50%)	6 (60%)	**10 (100%)**
Mean runtime per instance (s)	3,259.1	2,922.4	2,373.9	**924.4**	1,883.9

F.3 Results on Hard32-Fea

Table 2 and Section 4.4 give the feasibility results. This subsection adds how each system fails, solution quality, and runtime.

The systems fail differently. All 6 of DecisionBrain's non-passes are task-budget exhaustions with no candidate. Famou returns 23 candidates, of which 3 violate a constraint (two on time windows and depot return, one using three vehicles where two are allowed). Of its 9 runs without a candidate, 1 declares the instance infeasible without a schedule and 5 hit the task limit. COOPA returns 19 candidates and 14 fail: 9 leave requests or customers unserved (its only VRPTW candidate serves none of the 800 customers), 2 break time windows, 1 uses an extra vehicle, 1 exceeds the deadline, and 1 covers none of the 50 jobs. Rerunning 23 instances with a raised iteration cap added no feasible decision. OptiMUS's 2 candidates serve no request, its model having left route arcs disconnected from assignments so that a zero objective was reported optimal. OR-LLM-Agent returns no candidate. The 9 runs its adapter marked successful carried no objective value.

Solution quality. DecisionBrain's mean (median) gap against the reference method is $+0.0577$ ($+0.0355$) on job-shop, -0.0043 (-0.0003) on VRPTW, and $+0.0400$ ($+0.0139$) on pickup-and-delivery.

Runtime. In Table 17, DecisionBrain's mean over all 32 instances is raised by its 6 runs at the task limit. Its median and its mean over solved instances describe performance on solved instances. COOPA's low solved-instance mean covers 5 instances, four of them small job-shop instances. OR-LLM-Agent is fastest, stopping early without a candidate, and Famou's times include browser-automation overhead.

Table 17: Runtime over Hard32-Fea. Runs that exhausted the 7,200-second task limit are included at their measured duration, except one Famou run that overran the limit by more than 900 seconds and is counted at it. No COOPA run reached the task limit.

Metric	DecisionBrain	COOPA	Famou	OptiMUS	OR-LLM-Agent
Mean runtime (s)	2,960.9	3,106.5	2,972.5	3,000.6	**970.4**
Median runtime (s)	1,837.2	3,264.2	2,051.6	2,943.8	**881.1**
Mean, constraint-feasible instances only (s)	1,982.1	**1,650.9**	1,943.7	—	—

F.4 Ablations

The arms are run on Hard32-Fea and nested as described in Section 4.5. Each differs from its reference in exactly one mechanism. The subsections follow this nesting, so the library ablation comes first.

Arm configurations. Table 18 lists the three ablated settings as recorded in the run manifests. The model backend, task and solver budgets, intake mode, retrieval machinery, input schema, problem contract, and cross-package composition are identical in all arms. The design stage and its component composition are withdrawn together, as one mechanism, and replaced by a modeler that builds one monolithic model. Narrowing the pool removes the third-party solvers a component may call, not the retrieval machinery.

Table 18: Recorded configuration of each arm on Hard32-Fea. Differences from the arm each one is read against are set in bold.

Setting	Full pool	Gurobi only	One monolithic model	No review
Full algorithm package pool	on	**off**	**off**	**off**
Design stage	on	on	**off**	on
Feasibility review and repair	on	on	on	**off**

F.4.1 Removing the Algorithm Library

The arm narrows the pool to Gurobi. The restriction is enforced at directory, contract, and execution level, so no other package can be retrieved, named in a design, or imported.

Table 19 reports the outcome. Eleven instances are solved by both arms and six by neither. Under the full pool the accepted decisions run through family-specific methods (Section G.1). Under the narrowed pool every one runs through Gurobi's MILP solver.

In both arms every failure to pass (6 and 21) is exhaustion of the task budget without a candidate.

Solution quality. On the 11 instances both arms solve, the full pool has the smaller gap on all 11 (mean $+0.0678$ against $+0.2408$).

Table 19: Removing the algorithm library on Hard32-Fea, using the metrics in Table 2. The two arms differ only in the package pool. Neither arm had a candidate rejected, so that table's violation row is omitted here.

Metric	Full pool	Narrowed to Gurobi
Constraint-feasibility rate	**81.3%**	34.4%
Rate among completed candidates	100.0%	100.0%
Job-shop deadline ($n = 6$)	**6**	3
VRPTW min-fleet ($n = 6$)	**5**	0
PDPTW fleet-cap ($n = 20$)	**15**	8
Constraint-feasible candidates	**26**	11
Instances solved only by this arm	**15**	0
No-candidate instances	**6**	21

Runtime. Table 20 reports elapsed time. On the 11 paired instances the full pool is faster on 10, averaging 2,240.8 seconds against the same 3,621.6. The narrowed arm solves nothing outside that set, so its two figures coincide.

The narrowed arm takes more attempts rather than longer ones: 56 responsibility-directed repairs against 11, on 25 instances against 6.

Table 20: Runtime of the library ablation on Hard32-Fea, using the metrics in Table 17. Runs that exhausted the 7,200-second task limit are included at their measured duration.

Metric	Full pool	Narrowed to Gurobi
Mean runtime (s)	**2,960.9**	5,972.2
Median runtime (s)	**1,837.2**	7,201.3
Mean, constraint-feasible instances only (s)	**1,982.1**	3,621.6

F.4.2 Replacing the Design Stage by One Monolithic Model

A dedicated modeler writes one complete Gurobi model, as mainstream LLM-based optimization systems do, and SOLVING may only translate, execute, and repair it. Pool and FEASIBILITY REVIEW are the same in both arms.

Table 21 reports the outcome. Neither arm submitted a decision the checker rejected, so every non-pass in both is a run that reached the task budget with no candidate.

The loss is family-dependent: the monolithic model arm leads on job-shop, 5 against 3, and trails on pickup-and-delivery, 3 against 8 (Section G.1).

Solution quality. On the 4 instances both arms solve, mean gaps are indistinguishable ($+0.0964$ against $+0.0969$).

Runtime. Tables 22 and 23 report time and errors. Twenty-four monolithic model runs reach the task limit against 21, and the design-stage arm records fewer stage errors on 31 of 32 instances.

Table 21: Replacing the design stage by one monolithic model on Hard32-Fea, using the metrics in Table 19. Both arms are restricted to the Gurobi pool and both retain FEASIBILITY REVIEW. Neither arm had a candidate rejected, so that table's violation row is again omitted.

Metric	Design stage	One monolithic model
Constraint-feasibility rate	**34.4%**	25.0%
Rate among completed candidates	100.0%	100.0%
Job-shop deadline ($n = 6$)	3	**5**
VRPTW min-fleet ($n = 6$)	0	0
PDPTW fleet-cap ($n = 20$)	**8**	3
Constraint-feasible candidates	**11**	8
Instances solved only by this arm	**7**	4
No-candidate instances	**21**	24

Table 22: Runtime of the design ablation on Hard32-Fea, using the metrics in Table 20. Runs that exhausted the 7,200-second task limit are included at their measured duration.

Metric	Design stage	One monolithic model
Mean runtime (s)	**5,972.2**	6,517.1
Median runtime (s)	7,201.3	7,201.6
Mean, constraint-feasible instances only (s)	**3,621.6**	4,455.4

Table 23: Time and process cost of the design ablation on the 4 instances both arms carry to a feasible decision, with the all-32 error counts alongside. Stage errors are the counts recorded across the workflow's stages.

Metric	Design stage	One monolithic model
Paired instances	4	4
Mean runtime, paired (s)	**2,820.3**	3,902.5
Median runtime, paired (s)	**2,919.5**	4,138.7
Mean stage errors, paired	**13.2**	37.0
Median stage errors, paired	**12.0**	36.0
Mean stage errors, all 32	**24.5**	62.0
Total stage errors, all 32	**784**	1,983
Instances with fewer stage errors	**31**	1

F.4.3 REMOVING THE FEASIBILITY REVIEW

The arm removes FEASIBILITY REVIEW and the repair loop it drives. Everything else, including the Gurobi pool, is unchanged.

Table 24 reports the outcome. The two arms reach the same count with the same family split but not the same instances: each solves 5 the other does not.

Of the 5 unreviewed candidates the hidden checker rejects, two are job-shop schedules that overrun their deadline and three are pickup-and-delivery route sets that violate the checker's constraints, one of them serving no request while reporting an objective of zero.

Solution quality. On the 6 instances both arms solve, mean gaps are $+0.3295$ with FEASIBILITY REVIEW and $+0.2119$ without, both far above the full pool's.

Runtime. In Table 25 the unreviewed arm is faster, and none of its runs reaches the task limit against 21. On the 6 common instances it takes 2,429.0 seconds against 3,766.0.

Table 24: Removing the feasibility review on Hard32-Fea, using the metrics in Table 19. Both arms are restricted to the Gurobi pool and both retain the design stage. The violation row of Table 2 is reinstated here, the unreviewed arm being the only arm on this suite whose candidates the checker rejects.

Metric	With review	Without review
Constraint-feasibility rate	34.4%	34.4%
Rate among completed candidates	**100.0%**	68.8%
Job-shop deadline ($n = 6$)	3	3
VRPTW min-fleet ($n = 6$)	0	0
PDPTW fleet-cap ($n = 20$)	8	8
Constraint-feasible candidates	11	11
Instances solved only by this arm	5	5
Candidates violating a constraint	**0**	5
No-candidate instances	21	**16**

Table 25: Runtime of the review ablation on Hard32-Fea, using the metrics in Table 20. Runs that exhausted the 7,200-second task limit are included at their measured duration.

Metric	With review	Without review
Mean runtime (s)	5,972.2	**3,082.1**
Median runtime (s)	7,201.3	**3,192.1**
Mean, constraint-feasible instances only (s)	3,621.6	**2,646.9**

G Extended Discussion

This section gives the run-record evidence behind Section 5: what the design stage and the algorithm library change in the accepted decisions themselves, how failures are shaped in each arm, how the three task families separate the mechanisms, and why the two open-source baselines return nothing on Hard32-Fea.

G.1 Run Records behind the Ablations

This subsection gives the run records behind the reading of Section 5: what changes when the design stage replaces one monolithic model, and what the library of methods adds once it does.

Design stage. Table 27 gives two effects. First, the distribution of errors shifts away from execution. With the design stage, half occur in design, and solving accounts for 336. Without it, solving accounts for 1,767 and is the dominant error site on all 32 instances. An error in design costs a revised document rather than a failed execution.

Second, the generated programs invoke the solver through a different computational structure. The monolithic model arm builds at most one model, and on 5 instances none. With the design stage every program builds at least one, and typically several restricted MILPs: per-route sequencing, per-cluster routing, fleet repartition, and residual assignment. Each has its own time limit and starts from the current solution.

The recorded designs replace a solve for the entire task with several restricted solves. Heuristic code produces the incumbent around which those sub-models are posed. Across 73 solving passes on FrontierOR65-Fea, 67 carry a portfolio of two or three alternative routes before any is written.

On FrontierOR65-Fea the difference appears in quality rather than in reach. On the 21 instances whose program wraps construction and improvement search around the solver call (Table 26), DecisionBrain's mean gap is below every baseline's, with a pairwise record of 30 better, 9 equal, and 7 worse. For the remaining pairings, the counts in the same better/equal/worse order are 30, 32, and

10. Medians coincide at zero on three pairings, so the separation lies in the tail, and the relation is correlational: the stage adopts a search structure where the instance admits one.

Table 26: Solution quality on FrontierOR65-Fea, restricted to the 21 instances whose generated program wraps a construction and an improvement search around the solver call. Each pairing keeps the instances both systems solve feasibly and for which both report a gap, so these counts are subsets of the suite-wide paired counts in Table 14. Gaps are signed against the reference method in the direction of each instance's objective, so lower is better.

Comparison	Paired, of 21	DecisionBrain	Other system	Better/equal/worse
versus COOPA	13	**−0.1676**	+0.0891	9 / 3 / 1
versus Famou	11	**−0.1894**	−0.0480	6 / 2 / 3
versus OR-LLM-Agent	14	**−0.2193**	+2.5693	10 / 2 / 2
versus OptiMUS	8	**−0.2621**	+3.9550	5 / 2 / 1

Table 27: What the design stage changes on Hard32-Fea, read from the run records of the two arms of Section F.4.2. Both arms are restricted to the Gurobi pool. Errors are the recorded stage errors, attributed to the stage that raised them. The modeler stage of the right-hand arm is the stage the design stage replaces. The solver rows are read from the generated `solver.py` of each run.

Metric	Design stage	One monolithic model
Errors raised in the design stage	393	111
Errors raised in the solving stage	**336**	1,767
Errors raised in the review stage	**31**	84
Instances whose dominant error site is solving	**12**	32
Mean length of the generated program	862 lines	235 lines
Gurobi models built per program (mean)	2.9	0.8
Solver calls per program (mean)	3.5	1.1
Programs building more than one model	**25**	0
Programs building no model at all	**0**	5
Programs seeding a model from the current solution	**24**	0
Programs carrying bounds between solves	10	4

The gain in reach is family-dependent: 8 against 3 on pickup-and-delivery, but 3 against 5 on job-shop. One reading is that structure pays where no single model of the entire task is adequate and costs where one is, as for the standard disjunctive job-shop model. Six job-shop instances cannot settle this, and we record it as a hypothesis.

Algorithm library. With the full library, an instance is also reached in fewer passes. Table 28 counts stage re-entries, where the workflow rejects its result and tries again. There are 32 on 7 runs under the full pool against 130 on 25 runs under the narrowed pool.

The accepted decisions show why. Under the full pool job-shop decisions come from OR-Tools CP-SAT, VRPTW decisions from PyVRP's iterated local search, and pickup-and-delivery decisions from OR-Tools routing, with ALNS (Wouda & Lan, 2023) recorded on two and a dispatching rule on one job-shop instance. Each is the method that family is ordinarily solved by. The narrowed pool has one method for all three and rebuilds a MILP until one succeeds. On FrontierOR65-Fea, where a solver based on one monolithic model suffices, the library adds run time (Section 4.3).

G.2 The Shape of Failure

In both design-ablation arms every unsolved instance exhausted the task budget. No candidate was rejected and no run stopped early. Stage errors therefore measure effort, not soundness. The

Table 28: What the algorithm library changes on Hard32-Fea, read from the run records of the two arms of Section F.4.1. A stage re-entry is a stage the workflow enters again after having completed it once.

Metric	Full pool	Narrowed to Gurobi
Stage re-entries, total	**32**	130
Runs with at least one re-entry	**7**	25
Median re-entries per run	**0**	5
Largest number on one run	**5**	9
Errors raised in the solving stage	**183**	336
Mean runtime (s)	**2,960.9**	5,972.2
Mean, constraint-feasible instances only (s)	**1,982.1**	3,621.6
Instances carried to a feasible decision	**26**	11
Distinct methods behind the decisions	**5**	1

monolithic model arm records about two and a half times the errors, yet it leads on job-shop. The unreviewed arm is the only one with unsound returned decisions, and it has the second-lowest error count of the four.

G.3 Separation across the Three Families

The three families show different empirical sensitivities to the mechanisms. VRPTW shows the clearest library effect, the pickup and delivery family shows the clearest difference between the design arms, and job-shop shows neither. The family rows, not the pooled rate, are therefore the ones to read.

G.4 Baseline Failure before the Budget

OR-LLM-Agent terminates before the budget on all 32 runs at a mean of 970.4 seconds with no candidate. OptiMUS terminates before the budget on 31 runs and reaches the limit once. It returns no candidate on 30 instances, and its 2 candidates are both rejected. On FrontierOR65-Fea, where a solver based on one monolithic model suffices, OptiMUS reaches 32.3% and OR-LLM-Agent 84.6%. On Hard32-Fea both still build one monolithic model, and when that route yields no decision within the solver budget neither pipeline turns to another route before returning.

H Algorithm Library Reference Tables

This appendix collects the two reference tables of the structured algorithm library described in Section 3.3: the semantic blocks and validation layers of a package guide, and the packages, solver summaries, interface profiles, and application domains registered in the library. The catalog exposes the solver summaries shown in the second table. Detailed invocation facts are returned only by the corresponding package and solver guides. Registered packages include Gurobi (Gurobi Optimization, LLC, 2024), OR-Tools (Perron & Furnon, 2025), PySCIPOpt (Maher et al., 2016) on SCIP (Bestuzheva et al., 2023), PyVRP (Wouda et al., 2024), and ALNS (Wouda & Lan, 2023).

H.1 Semantic Blocks of an Algorithm-Package Guide

Table 29: Semantic blocks in an algorithm package guide.

Layer or block	Information represented	Purpose
Common manifest envelope	Package identity, maturity, delivery form, summary, status, and source	Establishes the package-level discovery unit and prevents unvalidated packages from entering the catalog.

Table 29 (continued)

Block	Information represented	Purpose
Catalog solver summary	Solver or backend identifier and one-sentence purpose summary	Extends shallow discovery to solver level while withholding detailed parameters, calls, result semantics, and guarantee evidence.
Interface profile	Strict package-specific schema for the exposed solver or backend suite	Ensures that the declared interfaces are complete, non-duplicated, and consistent with the package integration.
Algorithm semantics	Applicable problem families, method type, exact or heuristic class, and optimality boundary	Provides the main semantic basis for shortlisting packages.
Capabilities and limits	Supported modeling features, unsupported constraints, warm starts, randomness, time control, and numerical restrictions	Connects problem requirements to package capabilities and states the boundaries of valid use.
Selection boundary	Prerequisites, suitable cases, rejection conditions, and evidence about applicable instance scale	Prevents selection based only on a package or method name.
Distribution and environment	package name, validated and compatible versions, import name, Python version, license, documentation, source, native extension, and validation environment	Binds API facts to a tested version and environment and distinguishes importability from licensed solving capability.
Solver/interface entry	Solver or backend identity, modeling entry point, method class, guarantees, and invocation semantics	Distinguishes the concrete interfaces and guarantees exposed by one package profile.
Execution and result	Imports, construction and solve calls, key parameters, stopping conditions, result structure, and warnings	Constrains invocation, time control, termination interpretation, bound and gap semantics, and conversion into business outputs.
Agent guidance and acceptance	Design cautions, representative invocation, environment checks, and acceptance requirements	Gives the agent an executable reference and the conditions that must hold before relying on a selected interface.

H.2 Packages, Solvers, and Application Domains

Table 30: Packages, solvers, and application domains in the current library.

Package/version	Validated profile and solver/interface entries	Main domains	Key boundaries
Gurobi 13.0.2	`gurobipy`; `lp`, `milp`	General LP/MILP; assignment, packing, network flow, inventory, production planning, facility location, and linearizable routing	Exact models report incumbent, bound, and gap; a valid license is required, and the agent must still model variables, constraints, Big-M, and business output.
OR-Tools 9.12.4544	`ortools`; `cp_sat`, `linear_solver`, `min_cost_flow`, `routing`	Discrete scheduling and logic, assignment and packing, min-cost flow, routing, time windows, and pickup–delivery	Interfaces and guarantees differ across modules; CP-SAT requires integer semantics, while Routing usually returns a heuristic incumbent.

Continued on next page

Table 30 (continued)

Package/version	Validated profile and solver/interface entries	Main domains	Key boundaries
PySCIPOpt 6.2.1	`pyscipopt`; `milp`, `minlp` (plugin building blocks)	General MILP/MINLP; scheduling, routing, packing, cutting stock, and custom decomposition with SCIP plugins	Extensions are building blocks; a pricing extension does not create a restricted master, pricing problem, branching rules, or complete branch-and-price method automatically.
PyVRP 0.13.4	`pyvrp.ils`; iterated local search	CVRP, time windows, multi-depot and heterogeneous fleets, optional clients, multiple capacities, fixed vehicle cost, and multiple trips	A metaheuristic that proves neither optimality nor infeasibility; it supports neither arbitrary linear constraints nor general paired pickup–delivery precedence.
PyJobShop 0.0.9	`pyjobshop`; OR-Tools backend; CP Optimizer unavailable	Single and parallel machines; flow, open, job, and flexible job shops; project, multimode, and resource-constrained scheduling	Arbitrary constraints cannot be added outside its high-level interface; it does not cover general workforce law, crew pairing, or routing semantics.
JobShopLib 1.7.0	`job_shop_lib`; constraint programming, dispatching, annealing	Classical and flexible job shops; exact CP, fast construction, and simulated-annealing improvement	Guarantees differ; dispatching and annealing prove neither optimality nor infeasibility and do not claim general workforce or project scheduling.
RSOME 1.3.1	`rsome`; robust and distributionally robust models; eight deterministic adapters	Robust and distributionally robust optimization, uncertainty and ambiguity sets, linear decision rules, and deterministic reformulations	The agent must define uncertainty and sets; arbitrary stochastic decomposition is absent, and adapters differ in integer, conic, and proof-status support.
ALNS 7.0.0	`alns`; adaptive large-neighborhood search	Routing, resource scheduling, packing, cutting stock, inventory, and other problems with custom destroy/repair operators	Provides only a search framework. The agent implements the state, initial solution, objective, operators, and feasibility handling; the result is a heuristic incumbent.