Autonomous Engineering of the Physical World through Ontologies and Verifiable Optimization

Nexma Research[1]
Spatial Intelligence Group

Abstract

Engineering and operating the physical world - laying fiber and water networks, designing power grids and wireless coverage, routing delivery fleets, scheduling construction crews - is slow, costly, and bound to scarce domain experts. Recent advances in large language models have spurred interest in automating such work, yet current systems remain unreliable: asked for a concrete plan, a model produces output that is linguistically plausible but physically infeasible, violating capacity, budget, or geometric constraints in ways not apparent without re-deriving the result. We argue this gap is architectural rather than a matter of model scale, and that it is closed by separating two concerns. Generality is provided by an ontology - a schema-defined representation of a domain's entities, relationships, and constraints - operated by a single unmodified agent through a small set of generic primitives; switching domains requires loading a new schema, not modifying the agent. Feasibility is provided by verifiable optimization - each decision is formulated as a problem from a small family of classical optimization methods, solved deterministically, and validated against the domain's physical constraints before it is trusted. We present the resulting system, an agent that runs the full design loop across multiple physical domains, and the principles behind it. The contribution is a position and a systems architecture; we report illustrative behavior rather than benchmarks and omit implementation-specific tuning.

Abbreviations

MIP	Mixed-integer program	FTTH	Fiber-to-the-home
MILP	Mixed-integer linear program	GPON	Gigabit passive optical network
VRP	Vehicle routing problem	OLT	Optical line terminal
CP	Constraint programming	ONT	Optical network terminal
TSP	Traveling salesman problem	ILP	Integer linear program
MTZ	Miller–Tucker–Zemlin	GIS	Geographic information system
LP	Linear program	dB	Decibel

Contents

[1]Preprint. Implementation-specific tuning, dispatch policy, and prompt construction are omitted as proprietary. Figures cited in related materials are illustrative unless stated otherwise.

1 Introduction

Engineering and operating the physical world is among the most expensive activities a society undertakes, and among the least automated. Bringing a utility network, a transmission corridor, a coverage footprint, a delivery operation, or a construction programme from intent to execution is a sequence of slow, serial, expert-bound steps: a site is surveyed over months; a design or plan is drafted, checked against standards, and revised over quarters; only then does work reach the field. Each step depends on practitioners whose judgment is scarce and whose throughput is fixed. The cost is not only money and time but opportunity - the systems that most need building are gated behind a process that cannot scale. Although our worked example is a utility network, the same machinery applies wherever a physical decision must be made under constraint: routing vehicles, scheduling crews, placing equipment, analyzing flow.

Large language models have made the automation of this process newly plausible, because the human bottleneck is largely one of translation: turning an informal goal and a body of domain knowledge into a precise, constraint-respecting plan. Models are increasingly capable at exactly this kind of translation. Yet applied directly, they fail in a characteristic way. Asked to *describe* a design, a model is fluent and often correct; asked to *produce* one, it returns a result that reads as correct but does not survive verification - a run that exceeds its budget, an assignment that overloads a node, a route drawn through an obstacle the prompt forbade. These are not hallucinations of fact; the entities are real and the local reasoning is sound. They are constraint-satisfaction failures, and their defining property is that they are invisible to a reader who does not re-derive the arithmetic. A plan that has been read but not checked carries no guarantee of feasibility, and in physical engineering an infeasible plan is not a worse plan - it is not a plan at all.

We take the position that this gap is architectural, not a deficit of model capability, and that it will not be closed by scale. Feasibility is a property established by checking, not by fluency; a checked solution from a modest model is worth more than an unchecked one from a larger model. The productive question is therefore not how to build a model that designs infrastructure end to end, but how to factor the problem so that each part is handled by the mechanism suited to it. We identify two concerns that are usually conflated and argue they should be separated.

The first is *generality*: the demand that one system serve many domains - telecommunications, water, power, wireless, logistics - without being rebuilt for each. We obtain it from an ontology. A domain is described declaratively as a schema of entity types, the relationships among them, their properties, and the constraints that bind them; the agent operates this model through a small, fixed set of generic primitives, much as a developer operates any codebase through a handful of file operations. Domain knowledge lives in the schema and in the model's reasoning, not in bespoke per-domain tooling. Changing what the agent engineers is a matter of loading a different schema, not modifying the agent.

The second is *feasibility*: the demand that every decision the agent commits be provably consistent with the domain's physical limits. We obtain it from verifiable optimization. The agent recognizes which class of optimization a goal implies, formulates an instance from the ontology, dispatches it to a deterministic solver that returns a feasible - and where applicable optimal - result, and then validates that result against

the full constraint set, including physical checks the formulation itself may not encode. Nothing is rendered, surfaced, or acted upon until it has passed. When a check fails, its diagnostics drive a bounded revision and the loop repeats.

Combined, these two concerns describe an agent that runs the entire engineering loop - sensing, reasoning, designing, verifying, dispatching - across any physical domain its schema can express. The remainder of this paper develops the position and the architecture. Our contributions are:

- We frame the reliability of autonomous physical-world engineering as an architecture problem, and identify generality and feasibility as the two concerns to separate (Sections 2–3).
- We describe an ontology-based foundation, operated through generic primitives, that yields cross-domain generality without per-domain reengineering (Section 4).
- We describe a formulate–solve–validate loop over a small taxonomy of classical optimization families that yields feasibility by construction (Section 4).
- We present the integrated system as an autonomous engineer running the full loop, illustrate it end to end on a single domain, and discuss its limits honestly (Sections 5–6).

We report illustrative behavior rather than competitive benchmarks, and we omit the dispatch policy, numeric tuning, and prompt construction that make the system perform; these are noted as out of scope where relevant.

2 The Autonomous Engineering Problem

We begin by stating precisely the task the system is built to perform, since the choice of task determines what counts as success. We use the term *autonomous engineering* for the end-to-end production of a buildable design for a physical system from a high-level statement of intent, with no human in the formulation or solving loop.

Input. The input has three parts. The first is an *intent*: a natural-language statement of what is to be built - "serve these buildings with fiber," "bring water to this development," "cover this district above a signal floor." The second is a body of *geographic context*: the relevant physical state of the world, drawn from surveys, public geodata, or prior records - parcels, streets, existing ducts and poles, terrain, demand points. The third is a *domain specification*: the entity types, relationships, properties, and constraints that define what a valid design in this domain is. The first two vary per project; the third is the ontology, and it is what makes the task well posed - without it, "a valid design" has no meaning.

Output. The desired output is a complete design expressed in the terms of the domain specification: a set of typed entities with concrete positions and properties, a set of typed relationships connecting them, and the derived artifacts a builder requires - routes, schedules, material counts. The output is not advisory text about a design; it is the design itself, in a form a downstream crew or system can execute. Crucially, the output must be *feasible*: it must satisfy every hard constraint the domain specification declares.

What "feasible" means. Feasibility is the property that separates a design from a description. A design is feasible when (i) every entity respects its type's placement and capacity rules; (ii) every relationship is of an allowed kind between allowed endpoints and respects its own limits; and (iii) every constraint the domain declares - budgets, distances, depths, cascades, geometric exclusions - holds across the whole configuration. Feasibility is binary and checkable: a plan either satisfies the constraint set or it does not, and which constraints it violates can be named. This is what allows the output to be *verified* rather than merely reviewed.

Optimality versus feasibility. Most engineering problems admit many feasible designs; among them, some are cheaper, shorter, or more robust. Where a well-defined objective exists - typically cost - the system seeks not just a feasible design but a good one, and reports the optimality gap when the underlying solver provides it. But optimality is secondary to feasibility: a globally optimal design that violates a hard constraint is worthless, whereas a feasible design that is a few percent above optimal is deployable. The task is therefore: produce a feasible design, and, subject to feasibility, a near-optimal one.

Scope. The task as stated is the design loop - from intent to a verified, buildable design. Sensing (acquiring the geographic context) and dispatch (handing the design to field execution) bound the loop on either side and are treated as interfaces rather than contributions here. The domain specification is assumed given; authoring it is a separate activity, discussed only insofar as the system's generality depends on its existence.

3 Related Work

Our work sits at the intersection of three lines of research: the translation of natural language into formal optimization, the classical optimization of physical networks, and the construction of reliable agents over structured representations of the world.

Natural language to formal optimization. A body of recent work studies whether language models can turn a problem stated in words into an executable optimization model - defining decision variables, objective, and constraints, and emitting solver-ready code. The consistent finding is that the bottleneck is *formulation* rather than *solving*: the solvers are mature, and accuracy gains come from better translation, from structured multi-step prompting, and from feedback loops in which solver output is fed back to the model for revision. A second consistent finding is that formulation errors are *class-specific* - the mistakes a model makes formulating a routing problem differ from those it makes on scheduling - so guidance and checking are most effective when organized by problem class. We adopt both findings: our system treats formulation as the central difficulty, closes a feedback loop around the solver, and organizes guidance and validation by optimization family. We depart from this line in domain and grounding. Prior systems target optimization formulation in the abstract; we target spatial *engineering*, where the variables are geographic, feasibility includes physical and geometric constraints, and the output is a design that must render and build on real terrain. The formulation step is, for us, one stage inside a larger loop, not the whole task.

Classical optimization of physical networks. The optimization of infrastructure is a mature field. Network design problems reduce to well-studied combinatorial cores - facility location, network flow, Steiner trees, vehicle routing, constraint-based scheduling - each with provably correct exact methods and well-understood approximations. Fiber-network design is a representative case: it has been formulated as an integer linear program that jointly decides equipment placement and cable routing under capacity, distance, and tree-topology constraints, and, because the joint problem is NP-hard, addressed with model strengthening and multi-phase decomposition to keep large instances tractable. We do not contribute new algorithms to this literature. We treat its results as a library of verified primitives and place the burden of intelligence elsewhere: in deciding which formulation a stated intent implies, populating it from live geographic data, and checking the solved result against the domain's full constraint set - including physical constraints the formulation may not encode.

World models and agents over structured state. In much of modern AI a "world model" is a *learned*, predictive representation - a network that imagines future states in a latent space. Such models are powerful but opaque: their representation cannot be enumerated, and a candidate plan cannot be checked against an explicit constraint, because there is no explicit constraint to check. Our world model is of the opposite kind: an *explicit, typed* representation - entity types, relationships, and constraints, in the sense of a formal ontology - that the agent reads and writes and against which feasibility is defined. This explicitness is precisely what makes verification possible, and it is the reason an explicit, typed ontology, rather than a learned world model, is the right foundation for engineering. Separately, an emerging consensus on agent design favors small, generic, composable operations over large bespoke tools, and closes loops with verification rather than

trusting single-shot output. Our architecture is a spatial instantiation of that stance: the agent operates the ontology through a fixed, small set of generic primitives, and commits nothing it has not checked.

Geographic information systems. Conventional geographic information systems (GIS) are interactive tools: a human analyst loads layers, runs geoprocessing operations (buffer, overlay, network trace), and inspects results on a map. The intelligence is the analyst's; the software executes discrete commands. Our system is an *agentic* GIS in the sense that the analyst's role - deciding which operations to run, in what order, and judging the result - is performed by the agent, while the spatial data model and the operations remain those of a GIS. The agent reads and writes standard geographic features (points, lines, polygons in geographic coordinates), runs the same classes of spatial operations, and renders to the same kind of map; what changes is that a language model, not a person, drives the session, and that every result is checked against the ontology before it is kept. This positions the work as autonomous GIS for engineering, distinct from both human-operated GIS and from optimization systems that have no explicit domain model.

4 Architecture

The system is organized around the two concerns named in the introduction. Generality is supplied by an ontology and a small set of generic primitives (Sections 4.1–4.2); feasibility is supplied by a taxonomy of optimization families and the loop that wraps them (Sections 4.3–4.4). We describe each in turn, at the level of principle. Dispatch policy, numeric thresholds, and the formulation guidance that drives the agent are implementation-specific and omitted.

Figure 1 gives the architecture at a glance: intent enters at the top, the ontology and skill define the domain, the agent operates the Codex through generic primitives and the optimization families, and every committed write propagates reactively to the views.

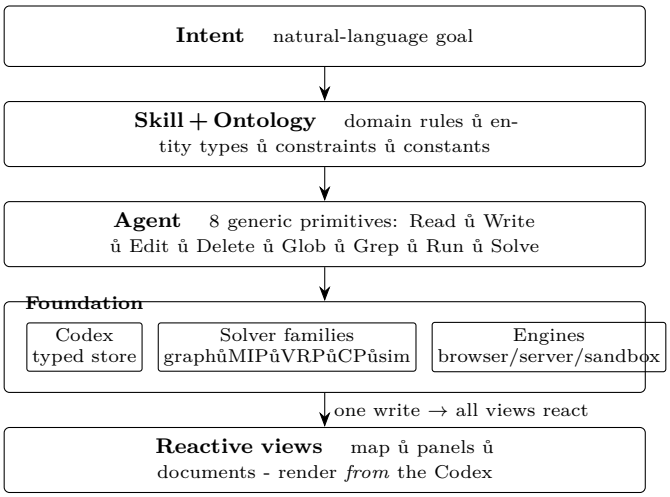

Figure 1: The system stack. The agent is constant across domains; the Skill + Ontology layer supplies the domain, and the reactive views are a pure function of the Codex - a single committed write updates map, panels, and documents at once.

4.1 The ontology

A domain is described declaratively as an *ontology*: a formal specification of the entity types that may exist, the relationships that may hold among them, the properties each carries, and the constraints that bind them. We use the term in its established knowledge-representation sense – a formal, explicit specification of a shared conceptualization of a domain [1] – and, because that specification is what the agent treats as its model of the world, we also call it the agent's *typed world model* where the contrast with the learned world models of Section 3 matters. The specification has four parts:

- **Entity types** - the nouns of the domain. Each declares a geometry (point, line, polygon), a tier in the domain's hierarchy, a set of typed properties, and placement rules (must it snap to a street; may it be underground; what is its capacity).
- **Relationship types** - the legal connections. Each declares which entity types it may join, in which direction, and its own constraints; an illegal connection is not representable.
- **Constraints** - the rules that make a configuration valid. Each is a typed predicate over entities and paths - a budget, a distance, a cascade depth, a geometric exclusion - with a severity (error or warning) and the parameters it references.
- **Constants** - the domain's named quantities, with units and a source standard, referenced by constraints and by the agent's reasoning.

This explicitness is what makes the rest of the system possible. Because every object is typed and every rule is named, a configuration can be *checked*: feasibility is defined relative to the specification, not left to judgment. And because a domain is *only* such a specification, changing domains is changing the specification, not the agent. The same machinery that engineers one domain engineers another whose ontology can be expressed in these four parts; in our deployment a range of domains across utilities, telecommunications, and operations are expressed this way, and the agent is common to all of them.

An ontology is authored and inspected visually as a typed graph: entity types are nodes carrying their properties, relationship types are the edges between them, and an inspector panel edits the selected element. Figure 2 sketches this view for the FTTH ontology.

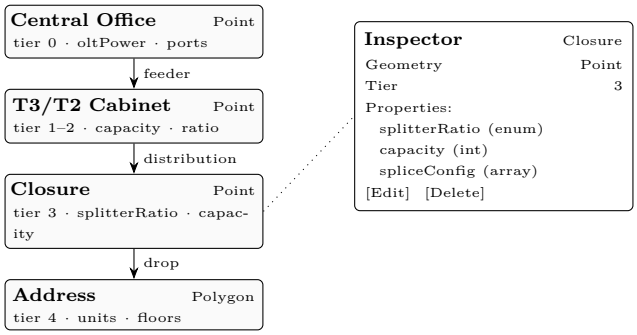

Figure 2: The ontology as authored on the `/ontology` canvas: entity types are nodes (name, geometry, tier, properties), relationship types are labeled edges, and an inspector edits the selected element. Editing this graph changes the typed world the agent reasons over; no agent code changes.

4.2 The Codex and the generic primitives

The ontology defines the *types*; the project's actual spatial state lives in a concrete, persistent store we call the *Codex*. The Codex is a database-backed virtual filesystem that holds the entire spatial state of a project - the typed entities and relationships, the ontology and skills that type them, live data feeds, the agent's own working notes, and the derived artifacts it produces. We name the agent that operates it *Jax*. The Codex, not the map or any panel, is the single source of truth: Jax reasons by reading it and acts by writing to it, and every visual surface is a function of it (Section 4.2.2).

Files are organized into namespaces by role, so that discovery is structural rather than guessed:

4.2.1 Generic primitives

Jax does not act through a catalog of domain-specific operations such as "place a closure" or "size a pipe." It acts through a small, fixed set of *generic primitives* over the Codex, in the same way a software agent operates any codebase through a handful of file operations: read a file, write a file, edit in place, search by content, list by pattern. To these file-like primitives we add two computational ones: a primitive for

```
/
  _meta/
    ontology.json                                  the ontology
    skills/{id}.md                                 agent capability packs
  feeds/{slug}.geojson                             live + imported data
  layers/{entity}.geojson                          the design Jax authors
  analyses/{slug}.geojson                          outputs of operations
  indexes/{slug}.idx                               spatial indexes
  zones/{id}/work-twin.txt                         text-twin grids (§4.7)
  exports/{slug}.{ext}                             field documents
```

Figure 3: The Codex namespace. Each role occupies a distinct path prefix; the agent discovers data by pattern rather than by assumption, and writes results to the namespace that matches their role. The ontology and skills live under `_meta/`; the design Jax produces accumulates under `layers/`.

deterministic geometric and numerical computation (distances, centroids, snapping, road-following routes), and a primitive for optimization dispatch, described in the next section.

The design rule is strict: the primitives are domain-neutral. A primitive that only makes sense for one domain is disallowed; such value belongs in the ontology, not in the tool set. Domain knowledge therefore lives in two places - the declarative specification, and the agent's own reasoning guided by it - and in neither case in bespoke tool code. The payoff is that the tool surface is constant across every domain: the agent that designs in one domain is, primitive for primitive, the agent that designs in another.

4.2.2 Reactive views and skills

Reads and writes to the Codex are bidirectional and incremental, so a design is built up, inspected, and revised in place. Because every visual surface is a function of the Codex, a single write propagates everywhere at once: when Jax writes a layer of placed closures, the map, the inspector, and any dependent analysis update without a separate rendering step. Jax never manipulates the map directly; it writes geography, and the views follow.

Capability targeting a particular ontology - Jax's instructions, which primitives it is permitted, and the validators that apply - is packaged as a *skill* bound to that model; one project binds one ontology and the skills that target it. The skill is how behavior is specialized without specializing the agent: Jax is constant, and the skill plus ontology supply everything domain-specific.

A skill is a structured document with named sections (Figure 4): *identity* (which ontology it targets, version), *capabilities* (toggles), the agent *instructions*, the permitted *tools* (a subset of the generic primitives), the *validators* that gate its output, and worked *examples*. Two projects on the same ontology can load different skills; the same skill is versioned independently of the ontology it targets.

Skill: FTTH Network Design	targets `ftth-gpon-v1` · v2.1.3
Identity	ontology, name, version, target platforms
Capabilities	toggles: optical-budget validation, splice generation
Instructions	the agent's domain brief (GPON rules, hierarchy, workflow)
Tools	Read · Write · Edit · Delete · Glob · Grep · Run · Solve
Validators	optical budget $\leq 28\,\mathrm{dB}$ · cascade depth ≤ 2
Examples	worked design traces (read-only)

Figure 4: A skill as authored on the `/skills` page: a sectioned document binding domain behavior to an ontology. The *tools* section grants a subset of the generic primitives; the *validators* section names the constraints that gate committed output. Swapping the skill changes behavior without touching the agent or the ontology.

Spatial state is stored as standard geographic features - points, lines, and polygons in the usual `[longitude, latitude]` order - so Jax works in a consistent rhythm: *discover* relevant data by pattern (Glob, Grep), *read* it to build context, *run* a spatial operation or a solve (Run, Solve), and *write* the typed result back as

a feature layer (Write). Operations are path-in, path-out: Jax passes Codex references rather than inlining large geometries, which keeps its working context small and its actions legible.

4.3 A taxonomy of optimization families

Feasibility is established by *mathematical optimization*. By this we mean the discipline of choosing decision values that minimize (or maximize) an objective subject to a set of constraints that the choice must satisfy - the formal core of operations research, with a body of exact and approximate algorithms developed and hardened over more than half a century. Its relevance here is specific: a spatial engineering decision is rarely a matter of taste but of finding the cheapest arrangement that violates no physical limit, which is exactly what an optimization solver computes. And it computes it with the two guarantees a buildable design requires and a language model alone cannot offer: a solution that provably satisfies every stated constraint, and, where an objective is defined, a bound on how far that solution sits from the best possible. Routing each decision to a solver is therefore not an implementation detail but the mechanism by which "feasible" acquires a precise, checkable meaning.

Every decision the agent commits is accordingly the solution of a constrained optimization problem of the canonical form

$$\min_{\xi \in \mathcal{X}} \; g(\xi) \quad \text{s.t.} \quad c_r(\xi) \leq 0 \;\; \forall r \in \mathcal{R}. \tag{1}$$

We read this expression term by term, since the same notation recurs throughout the paper. The operator min means "choose the values that make the following quantity as small as possible." The quantity being minimized, $g(\xi)$, is the *objective* – the cost of a design. Its argument ξ (the Greek letter xi) is the *decision vector*: the list of choices the design comprises – what to place, how to connect, when to act. The notation $\xi \in \mathcal{X}$ reads "ξ drawn from the set $\mathcal{X}$," where $\mathcal{X}$ is the space of structurally allowed decisions (for example, vectors whose entries are 0 or 1). The phrase s.t. abbreviates "subject to" and introduces the constraints. Each $c_r(\xi) \leq 0$ is one constraint: a quantity computed from the decision that is required not to exceed zero, indexed by r ranging over the constraint set $\mathcal{R}$; the symbol $\forall$ means "for all," so the line reads "for every constraint r in $\mathcal{R}$, the quantity $c_r(\xi)$ must be at most zero." Together: choose the cheapest decision among those that violate no constraint. The objective g and the constraints $\{c_r\}_{r \in \mathcal{R}}$ are read directly from the ontology.

What differs across problems is the structure of $\mathcal{X}$ and of the c_r: whether the variables are binary placements, edge selections on a graph, vehicle routes, or scheduled intervals. That structure is what selects a solution method. The system therefore organizes optimization into a small number of *families*, each a class of classical methods matched to a structural form of (1) and to a class of spatial decision. Table 1 lists them. The intelligence the system adds is not in the algorithms within each family - those are settled and provably correct - but in three steps the agent must perform: *recognizing* which family a stated intent implies, *formulating* a concrete instance from the ontology, and *validating* the solved result against the domain's full constraint set.

Family	**Decision it answers**	**Representative spatial use**
Graph	What connects, and along which path?	Shortest path, spanning / Steiner trees, flow
Mixed-integer	What to place and select, under budget?	Facility and equipment placement
Vehicle routing	In what order, by which vehicle?	Capacitated routing, time windows
Constraint prog.	When, on which resource, in sequence?	Scheduling, assignment, packing
Simulation	What happens, under physics?	Steady-state flow, signal propagation

Table 1: The optimization families. Each corresponds to a mature body of algorithms; the agent selects a family, formulates an instance from the world model, and validates the result.

Mixed-integer programming, in full. Because it is the family that does equipment placement – the workhorse of the worked example – we define it in detail, and in doing so introduce the matrix notation that the optimization literature uses for (1). A *mixed-integer program* (MIP) is the case of (1) in which the

objective and constraints are *linear* and *some* decision variables are required to take whole-number values while others may be continuous - hence "mixed." Its standard written form is

$$\min \{\, c^\top x \;:\; Ax \le b,\;\; x_j \in \mathbb{Z} \;\; \forall j \in \mathcal{I} \,\}, \tag{2}$$

which we read piece by piece. Here x is the decision vector - the concrete instance of ξ from (1) - a column of numbers, one per decision. The term $c^\top x$ is the objective: c is a column of cost coefficients, one per decision, and $c^\top x$ (read "c-transpose x") is their *dot product*, i.e. the weighted sum $\sum_j c_j x_j$ - the total cost of the decision x; the superscript $\top$ denotes transpose, the operation that turns the column c into a row so the multiplication yields a single number. The block $Ax \le b$ states all the linear constraints at once: A is a matrix of coefficients, b a column of bounds, and the product Ax a column of constraint values, each required to be at most the corresponding entry of b - one inequality per row of A. Finally $x_j \in \mathbb{Z}\ \forall j \in \mathcal{I}$ is the integrality requirement: $\mathbb{Z}$ is the set of integers (whole numbers), and the line reads "for every index j in the set $\mathcal{I}$, the variable x_j must be a whole number," while variables outside $\mathcal{I}$ stay continuous.

This integrality is exactly what makes MIP the right tool for placement: a closure is either opened or not (0 or 1), an address is either assigned to a given site or not, a cable count is a whole number - discrete choices a continuous optimizer cannot represent - while continuous variables carry quantities like flow or distance. The concrete placement instance, with its decision variables and each constraint written out, appears in Section 5.3, equations (4)–(8). MIP is mature and used across logistics, scheduling, and network design, and it is the engine behind both the fiber and water placement runs of Section 6.

Each family corresponds to established solver technology. The system exposes them behind a single problem contract - a uniform description of family, problem type, model, and hints - so that a formulated instance can be solved by whichever engine is appropriate without the agent knowing which. The choice of engine, and the thresholds that govern it, are an implementation concern and are not detailed here; the invariant we maintain is that correctness does not depend on where a problem is solved. Two standard techniques make the families practical at scale and are noted in principle: a fast heuristic pre-solve can supply an initial feasible solution that warm-starts an exact solver, and very large instances can be decomposed geographically - partitioned, solved, and merged - trading a bounded amount of global optimality for tractability. Where decomposition is used, the resulting design is feasible but not guaranteed globally optimal, and we say so rather than imply exactness.

Not every spatial operation is an optimization, and a disciplined system must know the difference. Many operations are deterministic geometry, retrieval, or transformation; these run through the computation primitive or native operations, which are cheaper, faster, and return results with a known schema. The optimization path is reserved for genuine decisions under constraint. Choosing the lightest tool that fits each step is itself part of the agent's competence.

4.4 The formulate–solve–validate loop

The two concerns meet in a loop that the agent runs for each decision:

1. **Recognize.** From the intent and the ontology, identify the optimization family and problem type the goal implies.

2. **Formulate.** Construct a concrete instance - variables, objective, constraints - populated from the ontology and expressed in the uniform problem contract.

3. **Solve.** Dispatch the instance to an appropriate engine, which returns a result with a status (optimal, feasible, infeasible, timeout), an objective value, an optimality gap where applicable, and diagnostics.

4. **Validate.** Check the result against the domain's constraints, *including* spatial checks the formulation may not have encoded - for example, that no realized path crosses an excluded region. Validation is the gate, not a formality.

5. **Commit or correct.** If validation passes, write the result to the Codex, where it becomes visible and actionable. If it fails, the diagnostics drive a bounded revision of the formulation, and the loop repeats.

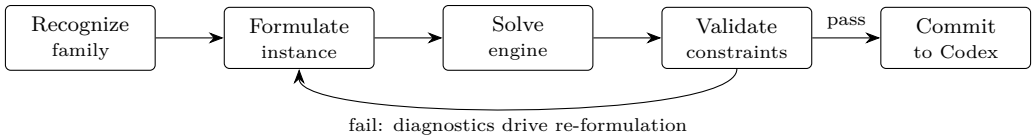

Figure 5: The formulate–solve–validate loop. A decision advances only when it passes validation; a failed check returns named diagnostics that drive a bounded re-formulation rather than a discarded result.

The box below (Algorithm 1) states the loop as pseudocode. It is the control flow, not the contents: the formulation guidance of Section 4.5 shapes FORMULATE and REFORMULATE, and is not reproduced here.

Algorithm 1. The formulate–solve–validate loop for one decision.
Input: intent q, ontology Ω, max correction rounds R.

1. $(family, type) \leftarrow \textsc{Recognize}(q, \Omega)$
2. $\mathcal{P} \leftarrow \textsc{Formulate}(family, type, \Omega)$ *// guided per family*
3. **for** $r = 1 \dots R$ **do**
4. $\quad (status, sol, diag) \leftarrow \textsc{Solve}(\mathcal{P})$
5. $\quad$ **if** $status \in \{\textsc{infeasible}, \textsc{timeout}, \textsc{error}\}$ **then** $\mathcal{P} \leftarrow \textsc{Reformulate}(\mathcal{P}, diag)$; **continue**
6. $\quad W \leftarrow \textsc{Validate}(sol, \Omega)$ *// incl. spatial checks beyond $\mathcal{P}$*
7. $\quad$ **if** $W = \varnothing$ **then** $\textsc{Commit}(sol, \Omega)$; **return** sol
8. $\quad$ **else** $\mathcal{P} \leftarrow \textsc{Reformulate}(\mathcal{P}, W)$
9. **return** $\textsc{report}(diag \cup W)$ *// no verified design; fail honestly*

Two properties of this loop are essential. First, it is *honest about failure*: an infeasible instance is reported as infeasible, with named diagnostics about which constraints conflict, rather than disguised as a confident but wrong plan. Second, it is *self-correcting*: validation failure is not a dead end but the input to the next formulation, so the agent converges on a verified design or terminates with a clear account of why none was found. The number of correction rounds is bounded. The loop is also observable: each step emits structured events, so a human can watch the agent recognize, formulate, solve, and validate - the spatial analogue of watching a coding agent run and read its own tests.

4.5 Formulation guidance

The hardest step in the loop is formulation. A capable model can still build a subtly wrong instance, and - importantly - the errors it makes are not random but *characteristic of the problem class*: the mistakes typical of a routing formulation (an incorrectly applied subtour-elimination constraint) differ from those of a flow formulation (a reversed conservation sign) or a scheduling one (a missing precedence link). Because these error modes are predictable per class, the system attaches class-specific *formulation guidance* when it formulates an instance: short, targeted modeling cautions keyed to the problem's structural family, drawn from a curated library and injected only when the matching family is active. The guidance does not solve the problem; it steers the model away from the recurring pitfalls of that family before the instance is ever sent to a solver, and it pairs with the family-specific validation that catches whatever slips through.

A canonical example makes the mechanism concrete. In the traveling-salesman and tree-routing families, a frequent failure is the Miller–Tucker–Zemlin subtour-elimination constraint applied to *every* node including the root: the position variable of the root is over-constrained, and the model is reported infeasible or yields a malformed tour. The corresponding guidance states the fix - fix one node's position and apply the constraint only to the others:

Recurring error (routing family). Subtour elimination applied to the root node ⇒ infeasible / malformed tour.

Guidance. Fix one node's position ($u_0 = 0$) and apply MTZ only to the *remaining* nodes:

$$u_i - u_j + n\, z_{ij} \le n - 1 \qquad \forall\, i, j \neq \text{root},\ i \neq j.$$

This is textbook operations research, reproduced here to show the form of the guidance, not its contents; the production library spans every family and is not disclosed. How a given problem *receives* the right guidance is itself part of the ontology: each skill declares the structural *archetypes* its domain exhibits - a fiber or water network declares `flow-node`, a patrol or collection route declares `route-sequence`, a coverage problem declares `coverage-emission` - and the matching guidance is drawn from a shared library keyed by those archetypes. A domain therefore inherits the accumulated formulation expertise of its archetypes without any of it being written into the agent: add a skill that declares `flow-node`, and its instances are guided by the same flow-conservation and subtour cautions that guide fiber and water, because they share the structure those cautions address.

Guidance also includes *valid inequalities* - constraints that are mathematically redundant (they exclude no feasible integer solution) but that tighten the linear relaxation a solver explores, often dramatically reducing solve time. A concrete example arises in fiber-network flow. Because fiber is laid in fixed-capacity cables, the fiber count w_e on an edge is always a multiple of the cable capacity κ whenever any cable is present ($a_e \ge 1$), so the inequality

$$w_e \;\ge\; \kappa\, a_e \tag{3}$$

holds at every integer optimum yet sharply cuts the fractional space between them.

Proposition 1. *A* valid inequality - *one satisfied by every integer-feasible point of a mixed-integer program but violated by some point of its linear relaxation - leaves the set of integer optima unchanged while strictly shrinking the relaxation.*

Proof. If every integer-feasible point satisfies the inequality, the integer-feasible set is unchanged, so the set of integer optima is unchanged. Since some relaxation point violates it, the relaxation's feasible region strictly contracts. Hence the inequality can only tighten the bound the solver works against, never exclude an optimal design. □

The fiber-cable bound (3) is such an inequality for this problem class. Because fiber is installed only in whole cables of capacity κ, a cost-minimizing design never buys a partially-needed cable, and the fiber and cable counts on an active edge line up at a whole-cable boundary; the bound encodes that structural fact, which holds at every integer optimum yet cuts fractional points of the relaxation between them. Including it preserves the optimal design while materially accelerating the search. The full library of such guidance and inequalities is organized by family and is implementation-specific; (3) is one illustrative instance of the kind, not the whole.

Figure 6 shows how guidance and validation compose into the self-correcting loop: a formulated instance is solved, the result is validated, and a failure feeds named diagnostics back into a guided re-formulation rather than being discarded.

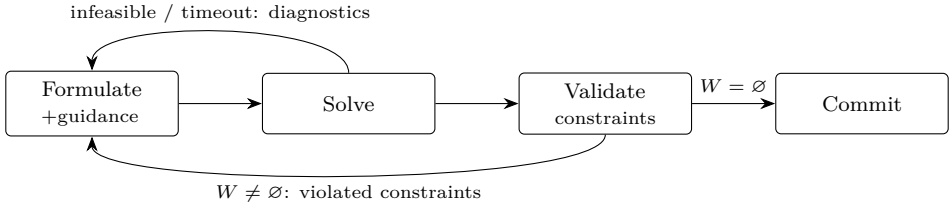

Figure 6: The self-correcting loop. Two distinct failure channels return to formulation: a solver failure (top, with solver diagnostics) and a validation failure (bottom, with the named violated constraints). Only a result that both solves and validates is committed.

4.6 Spatial comprehension

A language model formulates well only if it can perceive the geography it is reasoning about. Feeding raw coordinates into the context is both unwieldy and poorly suited to a model's strengths; feeding an image requires a vision pathway and still resists precise reference. The system instead gives the agent a *textual rendering* of space: each region of the working area is rendered as a compact character grid in which symbols denote the entities present - buildings, roads, placed equipment, endpoints - at a fixed spatial scale per cell. The agent reads this grid the way it reads source code, reasoning about adjacency, gaps, and whether a placement makes geographic sense, without a tool call and without leaving the text modality it is strongest in. The grids are generated deterministically from the Codex, so they are always consistent with ground truth, and they carry adjacency metadata linking neighboring regions. Figure 7 shows the form.

```
Zone A1 (200m × 200m, ∼20m/cell):
| ........  #### ...........    |    # = building
| .[].  #### ..####...    |          . = road
| ....  ....  ..####...    |         [] = placed equipment
| .......  0 ...........    |       @ = closure
| ..@.........  o .....    |        o = address
```

Figure 7: A textual rendering of one region (transliterated to ASCII for print). The agent perceives geography as a token grid generated deterministically from the Codex, enabling spatial reasoning within the text modality rather than over raw coordinates or pixels.

This textual rendering is what operationalizes the "third modality" of the introduction. Spatial reasoning becomes legible to the model not by enlarging the model but by presenting space in the representation the model handles best, and by keeping that representation a deterministic projection of the same ontology and Codex everything else reads.

5 Worked Example: Fiber-to-the-Home Network Design

To make the architecture concrete we trace it through a single domain in depth: the design of a fiber-to-the-home (FTTH) access network on a passive optical (GPON) architecture. We choose this domain because it is demanding - it couples equipment placement, cable routing, a hard optical budget, and field-level documentation - and because it lets us show that a domain-agnostic agent produces output at the depth a fiber engineer expects. We state the framing in advance: *none of the FTTH-specific content below is built into the agent.* Every entity type, every constant, every constraint named in this section is a declaration in the FTTH ontology. The same agent, given a different ontology, produces the analogous depth for a different domain; we return to this point in Section 5.10.

5.1 The FTTH ontology

The domain is specified as an ontology with a five-tier equipment hierarchy and four cable types that connect adjacent tiers. The equipment entity types are the *Central Office* (tier 0; the optical line terminal, origin of all fibers), the *T3 Cabinet* (tier 1; first-level distribution), the *T2 Cabinet* (tier 2; intermediate distribution), the *Closure* (tier 3; houses the splitter that fans fibers out to customers), and the *Address* (tier 4; the customer endpoint). Each declares typed properties and placement rules: a closure, for instance, declares a splitter ratio drawn from {1:4, 1:8, 1:16, 1:32}, a capacity of at most 32 served addresses, and a placement rule requiring it to snap to a street with a minimum spacing from its neighbors.

The cable types are connector entities, each legal only between specific tiers: a *feeder cable* (Central Office to T3, 48 fibers standard, maximum 2000 m), a *trunk cable* (T3 to T2, 24 fibers, maximum 500 m), a *distribution cable* (T2 to closure, 12 fibers, maximum 150 m), and a *drop cable* (closure to address, one fiber, maximum 100 m with 30 m recommended). Because these connectors are typed by endpoint, an illegal connection - a drop cable between two cabinets, say - cannot be represented. The ontology also admits a *conduit* entity for existing underground duct, which functions as a preferred cable path; we use it in Section 5.4.

The domain's constraints are typed predicates over this structure. The governing one is the *optical budget*: total optical loss from the Central Office to any address must not exceed 28 dB, the GPON Class B+

limit. Others bound the splitter *cascade depth* to at most two levels with a combined ratio no greater than 1:64, and cap each cable type at its maximum length. Each constraint carries its parameters and a standard reference; the constants they draw on - fiber attenuation of 0.35 dB/km at 1310 nm, 0.5 dB per connector pair, 0.1 dB per fusion splice, splitter losses of 7.3, 10.7, 14.1, and 17.5 dB for the four ratios, and a 3 dB contingency margin - are named quantities in the ontology, each with a unit and a source in ITU-T G.984 or TIA-598. Nothing in this paragraph is in the agent; all of it is in the specification the agent reads. Figure 8 shows the tier hierarchy and the cable types that connect adjacent tiers.

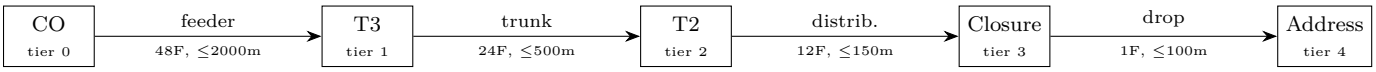

Figure 8: The FTTH ontology's equipment hierarchy and the typed cable connectors between tiers. Fiber counts and length limits are declared constants of the world model; a connector is legal only between the tiers shown.

5.2 Recognition and decomposition

Given the intent - serve a set of buildings from a source location - and the geographic context - the buildings, the street and duct network, the source - the agent recognizes that the task does not map to a single optimization but to a coupled sequence, exactly as the classical formulation of this problem does. There is a *placement* decision (where to put closures and cabinets, under capacity and reach limits), which is a mixed-integer problem with a facility-location core, and a *routing* decision (how to connect the placed equipment along legal, low-cost paths), which is a graph problem on the street and duct network. The hierarchy induces a natural decomposition, solved bottom-up: addresses are grouped into closures, closures into cabinets, cabinets to the source. For large service areas the agent decomposes geographically as well, partitioning into zones, solving each, and merging - accepting bounded sub-optimality at zone seams in exchange for tractability.

Spelled out, the design the agent must produce makes the following decisions while respecting the following rules, each of which is a typed constraint read from the ontology rather than logic written into the agent:

- Each address is served by exactly one closure.
- An address may be served only by a closure within the maximum drop distance (100 m; 30 m recommended).
- A closure serves at most as many addresses as its splitter ratio permits (eight at 1:8), and the ratio is itself chosen by the design.
- Each closure is connected to exactly one parent cabinet along a legal path no longer than the distribution limit (150 m).
- Each cabinet is connected upward to a parent cabinet or to the Central Office, within the feeder limit (2000 m).
- Cable is routed only along the street-and-duct network, never through a building footprint, preferring existing conduit where it exists.
- Total optical loss on every Central-Office-to-address path stays within the 28 dB budget, and no path exceeds two splitter stages.

The objective, subject to all of the above, is to minimize total installed cost: equipment plus cable plus civil works. Placement and routing, below, are the two optimization problems this decomposes into.

5.3 Placement

Closure placement is formulated as a capacitated facility-location instance. The candidate positions are points sampled along the routing network (so that a placed closure is, by construction, on a legal location);

the demand points are the addresses; the capacity of a closure is set by its splitter ratio, bounded by the ontology at 32 served addresses; and the objective is to minimize installed cost subject to covering every address within the drop-distance limit. The splitter ratio is itself a decision: the formulation may select 1:8 for a small cluster or 1:32 for a dense one, choosing the smallest ratio that covers the cluster so as to spend the least optical budget. The instance is dispatched to a mixed-integer engine, which returns the opened closures, their ratios, and the address-to-closure assignment, with an optimality gap. Cabinet placement repeats the pattern one tier up, with closures as the demand points and cabinet fiber capacity as the limit.

Concretely, let I be the set of addresses and J the candidate closure sites sampled along the routing network. With f_j the installed cost of a closure at site j, c_{ij} the cost of serving address i from j, κ_j the capacity implied by the chosen splitter ratio, d_{ij} the drop distance, and D the drop-distance limit from the ontology, the placement decision is the capacitated facility-location program

$$\begin{aligned}
\min_{x,y} \quad & \sum_{j\in J} f_j\, y_j \;+\; \sum_{i\in I}\sum_{j\in J} c_{ij}\, x_{ij} && && (4)\\
\text{s.t.} \quad & \sum_{j\in J} x_{ij} = 1 && \forall\, i \in I && (5)\\
& x_{ij} \le y_j && \forall\, i \in I,\ j \in J && (6)\\
& \sum_{i\in I} x_{ij} \le \kappa_j\, y_j && \forall\, j \in J && (7)\\
& x_{ij} = 0 \quad \text{whenever } d_{ij} > D && \forall\, i \in I,\ j \in J && (8)\\
& x_{ij},\, y_j \in \{0,1\}, &&
\end{aligned}$$

where $y_j = 1$ opens a closure at site j and $x_{ij} = 1$ assigns address i to it. Constraint (5) serves every address exactly once, (6) forbids assignment to an unopened site, (7) enforces splitter capacity, and (8) respects the reach limit. Both the parameters and the limit are read from the ontology; the agent *constructs* this program from the ontology rather than carrying it as fixed code, which is why the identical formulation, with κ_j reinterpreted as a pump or transformer capacity, places equipment in another domain.

5.4 Routing

With equipment placed, the agent routes cable as a graph problem over a *routing graph* built from the street and duct network. Edges carry costs that reflect real installation economics: a baseline for new sidewalk trenching, a steep multiplier for road crossings (permits, traffic, restoration), and a strong discount for routing through existing conduit, since reusing duct avoids excavation entirely. These multipliers are constants in the ontology, not hard-coded in the agent; the conduit entity declares its own routing effect. Distribution and feeder cable are then routed as minimum-cost trees connecting each cabinet to its children, with the graph's costs steering the solution onto existing infrastructure where it exists and minimizing new civil work where it does not. The realized cable paths, with their physical lengths, are written back to the ontology as typed connector entities.

Formally, the routing network is a graph $G = (V, E)$ whose vertices include the placed equipment and sampled path points, and the cable serving one cabinet and its children is a minimum-cost tree spanning that terminal set $T \subseteq V$. With a binary z_e selecting edge e, the decision is the Steiner-tree program

$$\min_{z} \quad \sum_{e\in E} c_e\, z_e \qquad \text{subject to the selected edges connecting all terminals in } T, \tag{9}$$

where the edge cost decomposes as

$$c_e \;=\; \rho_{\tau(e)}\, \ell_e, \tag{10}$$

the product of the segment length ℓ_e and a medium multiplier $\rho_{\tau(e)}$ keyed to the edge's installation type $\tau(e)$. The multipliers are world-model constants - existing conduit is strongly preferred over new trenching, and road crossings are penalized - so the same objective steers cable onto reusable infrastructure without any of those preferences being hardwired into the agent.

In practice the connectivity requirement over T is imposed by a *single-commodity flow* formulation, the operational realization of the Steiner-tree objective (9). A continuous flow variable $f_e \ge 0$ accompanies each

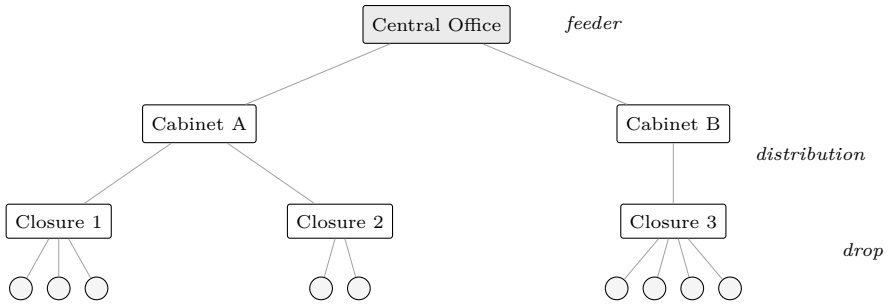

Figure 9: The committed FTTH design as a rooted tree. Leaves are served addresses; each closure rolls its demand up to a parent cabinet, each cabinet up to the Central Office. Fiber accounting (Algorithm 2) is a single leaf-to-root pass over this tree: the count of fibers a segment must carry equals the number of addresses in the subtree beneath it. The same structure is what the splice schedule and the bill of materials are computed from.

edge; the root injects one unit of demand for every terminal, each terminal absorbs one unit, conservation holds at transshipment nodes, and flow is permitted only on selected edges via the coupling $f_e \le M z_e$. Because every selected edge carries cost, a minimum-cost solution that satisfies this flow leaves no cycles - the selected edges form a tree. An equivalent encoding removes cycles explicitly with Miller–Tucker–Zemlin (MTZ) subtour-elimination constraints applied to every node but the root,

$$u_i - u_j + n\, z_{ij} \;\le\; n - 1 \qquad \forall\,(i,j) \in E,\; i,j \neq \text{root}. \tag{11}$$

Here n is the number of nodes, z_{ij} is the binary variable selecting the edge from node i to node j (as in (9)), and u_i is an auxiliary *position* variable assigned to each node: an ordering label that increases along the tree away from the root. The inequality forces these labels to increase across any selected edge, which is impossible to satisfy around a closed loop - hence no cycles can form. The constraint is applied for every edge (i,j) in the edge set E except those touching the root, whose label is fixed; omitting that exclusion over-constrains the root and is the classic error the guidance of Section 4.5 targets. The choice between the flow and MTZ encodings, and the conservation signs and root exclusion they turn on, is exactly where formulation errors cluster.

5.5 Validation: the optical budget

Placement and routing produce a configuration that is feasible *in the formulation*, but the governing physical constraint - the 28 dB optical budget - spans the whole path from source to customer and must be checked on the realized design. For each address i the agent sums the losses along its path p_i from the Central Office and requires the total to stay within the budget B:

$$L_i \;=\; \alpha \sum_{e \in p_i} \ell_e \;+\; \sum_{k \in K(p_i)} \sigma_k \;+\; \gamma_c\, n_i^c \;+\; \gamma_s\, n_i^s \;+\; M \;\le\; B \qquad \forall\, i \in I, \tag{12}$$

where L_i is the total optical loss to address i; p_i is its path (the sequence of fiber segments from the Central Office to that address) and $\sum_{e \in p_i}$ sums over those segments; α is the fiber attenuation per unit length and ℓ_e the length of segment e, so $\alpha \sum_e \ell_e$ is the distance-dependent loss; $K(p_i)$ is the set of splitters encountered on the path and σ_k the loss of the k-th of them; γ_c and γ_s are the per-connector and per-splice losses, incurred n_i^c and n_i^s times respectively along the path; M is the contingency margin; and $B = 28\,\text{dB}$ is the GPON Class B+ budget. The relation $\le B$ for all i requires every address to stay within budget. Every coefficient in (12) is a named ontology constant; the same predicate, with $\alpha, \sigma, \gamma, M, B$ rebound, expresses a water network's pressure floor or a wireless link's margin. Instantiating it on a representative single-stage path - roughly 3.2 km of total fiber ($\alpha = 0.35\,\text{dB/km}$), a 1:32 closure ($\sigma = 17.5\,\text{dB}$), three connector pairs, and five splices - gives

$$0.35 \times 3.2 \;+\; 17.5 \;+\; 0.5 \times 3 \;+\; 0.1 \times 5 \;+\; 3.0 \;=\; 23.6\ \text{dB} \;\le\; 28,$$

comfortably within budget. The check is not a formality: it is the step at which a plausible-looking design is proven deployable or rejected, and it uses the world model's constants throughout, so the same routine validates any domain's path-budget constraint with that domain's numbers.

5.6 Self-correction

Validation earns its place when it fails. Consider a dense multi-dwelling cluster that the placement step proposes to serve by *cascading* two 1:8 splitters - a tempting choice, because it economizes on distribution fiber. The combined ratio, 1:64, is within the cascade limit, so the topology passes a naive check. But the aggregate splitter term $\sum_k \sigma_k = 10.7{+}10.7 = 21.4\,\text{dB}$ already exceeds the ontology's 21 dB cap on cumulative splitter loss, and once the remaining terms of (12) are added the path violates the 28 dB budget. Validation reports this with named diagnostics: the violated constraint, the offending path, and the computed value of L_i. Those diagnostics drive a bounded re-formulation - the agent replaces the cascade with a single 1:32 splitter at the closure, or splits the cluster across two closures fed by separate distribution fibers - and re-solves. The loop repeats until every address path is within budget. The design that is finally committed is not the first one the optimizer produced; it is the first one that passed.

5.7 Tree extraction and fiber accounting

Placement and routing fix *where* equipment sits and *how* cable runs, but the design is not yet buildable. A crew also needs to know, for every cable segment, how many individual fibers it must carry, and at every closure which incoming fiber is spliced to which outgoing one. Both follow from one structural fact: the committed entities form a rooted tree (Figure 9), with served addresses at the *leaves* (nodes with no children) and the Central Office at the *root*. The number of fibers a segment carries is exactly the number of addresses in the *subtree* beneath it - the set of all addresses reachable downward through that segment.

Computing these counts is a single pass over the tree, from leaves to root. The agent first reconstructs the parent-child topology from the committed connector entities: each cable names its two endpoints, so the cables induce a graph whose tree structure is recovered by walking outward from the root and orienting every edge toward it. It then processes nodes in *reverse topological order* - every child strictly before its parent, the ordering produced by Kahn's algorithm, which repeatedly removes nodes whose dependencies are already resolved - and accumulates demand upward: each leaf contributes one fiber, and each internal node's outgoing demand is the sum of its children's. The roll-up is the production algorithm in `FTTHNetworkV2.buildClosureTopology` and `calculateAllFiberDemands`; Algorithm 2 states it in the abstract.

Algorithm 2. Tree extraction and bottom-up fiber accounting.
Input: committed connector entities E (each naming endpoints u, v); root r (Central Office).
Output: fiber count $f[e]$ for every segment e; parent map $p[\cdot]$.

1. build an undirected graph from E; traverse from r, setting $p[c] \leftarrow$ parent of c for every other node
2. **for** each node v: $d[v] \leftarrow 1$ if v is a served address, else $d[v] \leftarrow 0$ *// demand*
3. count each node's children; queue $Q \leftarrow$ all leaves (nodes with no children)
4. **while** Q not empty **do**
5. pop v from Q; let e be the segment $v \to p[v]$
6. $f[e] \leftarrow d[v]$ *// fibers carried by this segment*
7. $d[p[v]] \leftarrow d[p[v]] + d[v]$
8. mark one child of $p[v]$ resolved; **if** all of $p[v]$'s children resolved **then** push $p[v]$ to Q
9. **return** f, p

No optimization is involved - the tree is already fixed by placement and routing - so this runs through the RUN primitive (deterministic computation) rather than SOLVE, in time linear in the number of entities.

The output - a fiber count on every segment and a parent on every node - is precisely what the splice schedule and bill of materials below are generated from: a segment carrying k fibers needs a cable of at least k strands, and a closure with k fibers entering and k leaving needs k splices.

5.8 Derived artifacts

A design is not finished when its topology is feasible; a crew needs documentation, and a buyer needs costs. From the validated ontology the agent derives these field-level artifacts directly, because the information they require is already typed and present. We describe the three that matter most.

The splice schedule. At each closure, fibers from the incoming cable must be physically connected - *spliced* - either to a splitter that fans them out to local customers or onward to a downstream closure. A *splice schedule* is the per-closure table a technician follows to make those connections: for each incoming fiber it records the fiber's color code (under the TIA-598 standard, which assigns a fixed color order - blue, orange, green, and so on - so that any two technicians identify the same fiber identically), its disposition (drop to a local address, or pass through to a downstream node), and the splitter port or onward fiber it joins. The schedule must *balance* at every node: incoming fibers equal those dropped plus those passed through plus a small reserve. Table 2 shows the form for an illustrative 1:8 closure. Because color, disposition, and topology are all typed properties in the ontology, the schedule is a direct read-out, not a separate computation; the same projection produces the analogous connection table for any domain whose connectors carry an identity and a disposition.

In fiber	Color (TIA-598)	Disposition	Joins
1	blue	drop	splitter port 1 → addr. A
2	orange	drop	splitter port 2 → addr. B
3	green	drop	splitter port 3 → addr. C
⋮	⋮	⋮	⋮
8	black	drop	splitter port 8 → addr. H
9	yellow	pass-through	downstream closure CL-7
10	violet	reserve	-

Table 2: Illustrative splice schedule for a 1:8 closure: eight fibers dropped to local addresses through the splitter, one passed through downstream, one held in reserve. Fiber accounting balances ($8 + 1 + 1$ in $= 8$ drop $+1$ pass $+1$ reserve). Colors follow TIA-598. Values illustrate the artifact's form.

The same splice data also renders as a *splice diagram*: a schematic of the closure's splice tray that a technician follows on site. Crucially, the diagram is not drawn by hand - it is generated from the committed geographic design. The placed closure and the cables routed into and out of it (Section 5.4) fix which fibers arrive at the tray, and the tree extraction and fiber accounting of Section 5.7 determine, from that spatial topology, which incoming fiber feeds the splitter, which splitter outputs drop to which addresses, which fibers pass through to downstream closures, and which are held in reserve. Figure 10 is the resulting diagram for the 1:8 closure of Table 2; change the design on the map - move the closure, re-route a cable, add an address - and the diagram regenerates to match, because it is a projection of the same world model, not a separate drawing.

Bill of materials and bill of quantities. The same committed design yields the project's costing document. A *bill of materials* (BoM) enumerates the discrete components a build consumes - so many closures of each capacity, splitters of each ratio, connectors, splice trays - a parts list independent of geography. A *bill of quantities* (BoQ) extends this with the *measured* work the design implies - metres of cable by tier, number of splices, the labor each entails - each priced by the ontology's unit costs to yield a costed estimate. The distinction is conceptual: the BoM is what a warehouse picks, the BoQ is what a project is budgeted and tendered against, and the system emits a single costed bill spanning both. In our deployment that document is organized into three sections - equipment grouped by entity type, cable grouped by tier, and labor grouped by task - and exported as a print or spreadsheet artifact. Table 3 shows its structure; every line item is a typed entity or realized cable length of the committed design multiplied by a declared unit cost, so the bill is, once more, a projection of the validated ontology rather than a separate computation.

The point of the derived artifacts is that they require no additional intelligence and no FTTH-specific code path: they are projections of a typed, validated world model, and the same projection mechanism yields

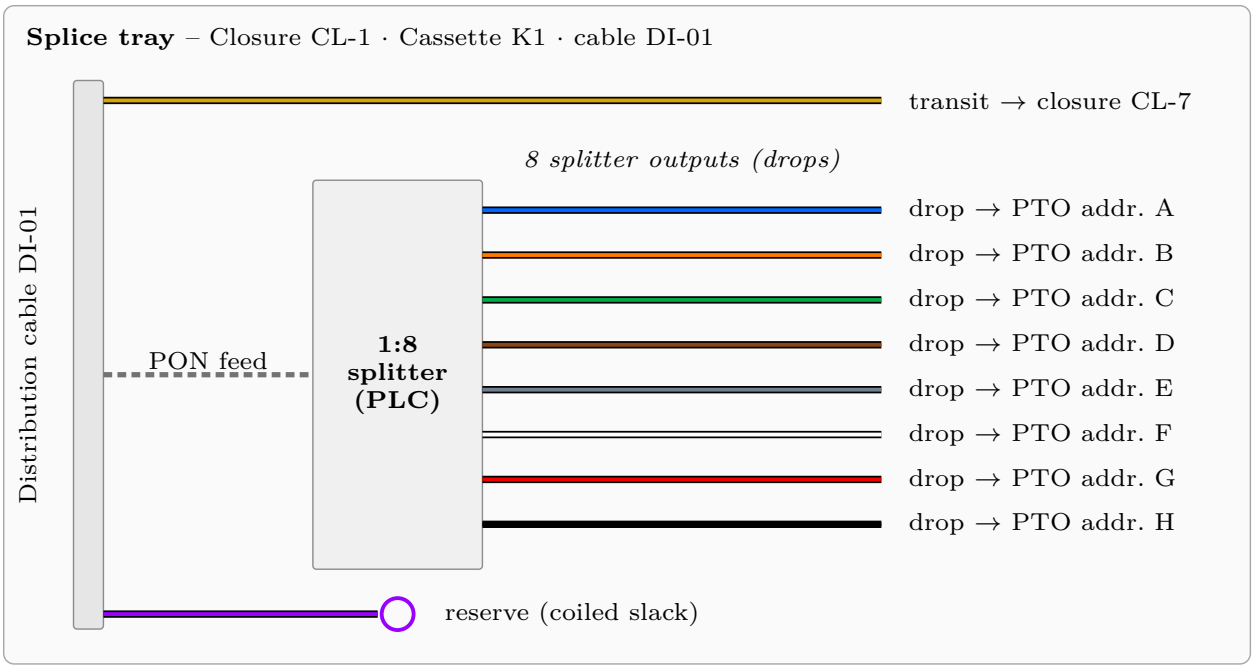

Figure 10: Splice diagram for a 1:8 closure, generated from the committed map design (the same splice configuration as Table 2). One PON feed fiber enters the splice tray and is fused to the input of a 1:8 optical splitter; the splitter's eight outputs are spliced to drop fibers serving customer terminations (PTO, the optical wall socket) at addresses A–H. One incoming fiber passes through untouched to a downstream closure, and one is coiled as reserve slack. Fiber colors follow the TIA-598 standard order. The diagram is a projection of the placed closure and routed cables, not a hand drawing.

the analogous documents for any domain whose ontology carries the relevant properties.

5.9 An interaction trace

To make the loop concrete, the box below traces a single intent through the primitives. It shows the *sequence of actions* the agent takes, not the prompts that produce them; the latter are implementation-specific.

User intent. *"Serve the buildings in this area with fiber from the exchange at the marked point."*

```
1  Glob "layers/*address*"        // discover the demand layer
2  Read "layers/addresses.geojson"        // build context
3  Read "_meta/ontology.json"     // confirm types, constants
4  Solve{family: mip, type: facility_location, ...} // place closures
5  Solve{family: graph, type: steiner_tree, ...}    // route cable
6  Run{type: optical_budget, path: ...}     // validate every leaf
7  budget exceeded on 3 paths → re-formulate (step 4) // self-correct
8  Write "layers/closures.geojson"    // commit; map reacts
9  Write "layers/distribution.geojson", drop.geojson
10 summarize: counts, total length, worst-case margin
```

The agent's behavior is governed by its active skill, whose instructions encode the domain's engineering rules. The sketch below is an *illustrative, abbreviated* rendering of such a skill's content for the fiber domain - enough to show the form; the production guidance, the formulation hints of Section 4.5, and the exact wording are not reproduced.

Section	Item	Unit	Unit cost (US$)
Equipment	8-port closure	each	800
Equipment	48-fiber cabinet	each	3,000
Cable	Feeder (tier 1)	per metre	370
Cable	Distribution (tier 2)	per metre	315
Cable	Drop (tier 3)	per metre	160
Labor	Fusion splice	each	50
Labor	Trenching (urban)	per metre	150

Table 3: Structure of the costed bill, organized into the system's three sections - equipment by entity type, cable by tier, labor by task. Unit costs are the world model's constants; quantities, omitted here, are read from the committed design. Values illustrate the artifact, not a costing of any specific network.

```
# Fiber Network Design (illustrative skill sketch)
You design GPON Class B+ access networks.
Hard constraints:  optical budget ≤ 28 dB OLT→ONT; cascade ≤ 2 stages;
                   drop ≤ 100m; distribution ≤ 150m; feeder ≤ 2000m.
Loss reference:  fiber 0.35 dB/km; connector 0.5 dB; splice 0.1 dB;
                 splitter 1:8 = 10.7, 1:16 = 14.1, 1:32 = 17.5 dB; margin 3 dB.
Hierarchy:  Central Office → Cabinet → Closure (splitter) → Address.
Workflow:  read the address layer; cluster into closures; place on streets,
           not in buildings; route along existing conduit where cheaper;
           compute the budget at every leaf and rework any path over 28 dB.
Operate only through the generic primitives; read types from the ontology.
```

Two things are worth noting about the sketch. First, every quantity in it - the 28 dB budget, the per-element losses, the distance limits - is a value the skill *references from the ontology*, not a number wired into the agent; swapping the ontology swaps these. Second, the skill specifies *what makes a design correct* and *the order of work*, but not *how to formulate* each optimization; that is supplied, per family, by the guidance of Section 4.5 and enforced, per family, by validation. The division is deliberate: domain rules live in the skill, formulation expertise in the guidance, and neither is hard-coded into the agent.

5.10 Generalization

Every step above - recognition, placement, routing, validation, correction, documentation - was performed by the domain-agnostic agent of Section 4 reading the FTTH ontology of Section 5.1. Nothing in the loop inspected a hard-coded notion of "fiber"; it inspected typed entities, named constants, and declared constraints. Replace the ontology and the same loop performs the corresponding work for a different domain: a water ontology substitutes pressure floors and pipe capacities for the optical budget and splitter ratios, and the placement, routing, validation, and bill-of-quantities steps proceed unchanged in structure.

This is not a thought experiment. Consider the placement step concretely in a second domain. A water distribution project replaces the address layer with a meter layer and the closure entity with a valve-station entity carrying a service-radius limit and a per-station capacity. Jax recognizes the same decision type - capacitated facility location - and constructs the *same* program of Section 5.3, with κ_j now a station's meter capacity and D now the service radius. The instance is dispatched to the *same* `mip/general` solver. No closure-specific or fiber-specific code participates; the only difference between the fiber run and the water run is the ontology each reads. Section 6 reports both runs, measured, side by side - the generality claim reduced to two rows of a table produced by one code path.

The depth demonstrated here is therefore not evidence of an FTTH system; it is evidence that the general system reaches domain-expert depth *through* a world model, which is the paper's central claim.

6 Evaluation

We report measured behavior of the system's solver layer across the optimization families and across domains. The aim is not a competitive benchmark - we do not compare against other systems on standard instance libraries - but a demonstration that the architecture's two central claims hold operationally: that one solver interface serves multiple physical domains unchanged (generality), and that the loop establishes and checks feasibility on realistic instances (feasibility by construction).

Setup. All runs invoke the production solver entrypoint directly - the same command the agent issues inside its sandbox - on representative instances of realistic scale. The instances are synthetic but dimensioned to real practice: geographic coordinates at metric scale, capacities and distance limits drawn from the relevant ontology. The solvers, the dispatch contract, and the result schema are the production ones; reported times are wall-clock on a single commodity core. The instances are not customer data, and the numbers are not offered as a benchmark against competing tools - only as evidence that the live system produces verified, optimal results on each family at realistic size.

Task	Family / type	Status	Objective	Time (ms)
FTTH closure placement	mip / general	optimal	US$298,965	104.8
Water valve placement	mip / general	optimal	US$142,287	40.7
Cable routing	graph / steiner_tree	optimal	850 m	3.2
Optical-budget check	simulation / rf_prop.	feasible	20.8 dB	1.0
Crew schedule	cp / scheduling	optimal	465 (makespan)	120.1
Technician tour	vrp / tsp	optimal	10,973 m	5089.4

Table 4: Measured runs of the production solver across five families on representative instances. Every instance returns OPTIMAL (or, for the simulation, a converged FEASIBLE analysis). Times are real wall-clock; solver engines are SCIP (mip), NetworkX (graph), OR-Tools CP-SAT (cp) and routing (vrp), and the signal-path analyzer (simulation). These are representative instances run through the live solver, not a competitive benchmark.

Generality, demonstrated. The first two rows are the load-bearing result. The FTTH closure placement (40 addresses, 12 candidate sites, capacity 8, 100 m drop limit) and the water valve placement (30 meters, 9 candidate stations, capacity 6, 150 m service limit) are *the same optimization*: both are issued to the `mip/general` solver and solved by the identical SCIP code path. Only the ontology differs - the FTTH instance carries splitter capacities and a drop-distance limit, the water instance carries station capacities and a service-radius limit. The FTTH run opened 11 closures and assigned 39 of 40 addresses (one address lay beyond 100 m of every candidate and was correctly left unassigned rather than forced into an infeasible drop); the water run opened 9 stations and served all 30 meters. Nothing in the solver, the contract, or the result handling was specialized per domain. This is the generality claim of Section 1 reduced to a reproducible measurement: change the ontology, not the code, and the same engine engineers a different domain.

Feasibility, checked. The optical-budget row exercises the validation step on a realized path. Running the path OLT $\rightarrow$ T3 $\rightarrow$ closure (1:32) $\rightarrow$ ONT through the signal-path analyzer accumulates 20.8 dB of loss - 1.9 dB on the feeder, 18.3 dB across the distribution stage (dominated by the 17.5 dB splitter), 0.6 dB on the drop - leaving a 7.2 dB margin against the 28 dB Class B+ budget; all three links pass. This is the same check that, on an over-cascaded design, returns a violation and drives the self-correction of Section 5.6. Feasibility here is not asserted from the topology; it is computed from the realized design and reported with its margin.

Coverage and cost. The remaining rows confirm the other families return optimal results at realistic size within interactive or near-interactive time: a 17-edge minimum-cost Steiner tree over a 36-node street graph in 3 ms; a precedence-constrained crew schedule (makespan 465) in 120 ms; an 8-stop technician tour (10.97 km) in about 5 s. The objectives are the quantities a planner acts on - installed cost in dollars, cable

metres, schedule makespan, travel distance - read directly from the committed result, which is also the basis of the bill of quantities of Section 5.8.

7 Discussion and Limitations

Why architecture rather than scale. The recurring temptation in this problem is to wait for a model capable enough to produce buildable designs end to end. We think this misreads where the difficulty lies. Feasibility is established by *checking*, not by fluency, and a checked design from a modest model is worth more than an unchecked one from a larger model. The separation of formulation from solving and validation is therefore not a stopgap until models improve; it is the correct factoring of a problem one part of which (interpreting open-ended intent) suits a language model, and another part of which (constraint satisfaction and optimality) suits a solver. As models improve, the formulation step improves with them - but the value of an explicit, checkable design does not diminish, because it is precisely what an infeasible-but-fluent plan can never offer.

Limitations. We name three honestly. First, *formulation remains the hard part*: the agent can still misrecognize an intent or build a subtly wrong instance. This is exactly why validation is non-optional and why the loop is self-correcting; the architecture contains the failure rather than eliminating it. Second, *decomposition trades global optimality for scale*: zone-partitioned designs are feasible and near-optimal, not provably globally optimal, and we report them as such. Third, *validation is only as complete as the ontology*: a constraint absent from the specification cannot be checked, so the quality of a domain's guarantees is bounded by the quality of its ontology. None of these is dissolved by a larger model; each is addressed by better formulation guidance, better decomposition, and richer specifications - and the first and third are made visible, not hidden, by the explicit-and-checkable design we advocate.

On evaluation. The measurements of Section 6 are representative runs of the production solver, not a competitive benchmark. We make this boundary explicit: the optimization engines are well characterized in their own literatures, and their performance on standard instance libraries is neither in question nor our contribution. What Section 6 establishes is narrower and, for our thesis, more to the point - that the same solver interface serves two different physical domains with no per-domain code, and that the loop returns verified, optimal results at realistic scale. A broader quantitative study - many instances per family, larger sizes, end-to-end agent runs measured against expert-produced designs - is future work. We report what we have measured and label its scope honestly rather than inflate representative runs into a benchmark.

8 Conclusion

Autonomous engineering of the physical world has been held back not by a shortage of model capability but by a missing architecture: language models can describe infrastructure fluently yet produce designs that fail on inspection, because nothing forces their output to satisfy the physical constraints that make a design buildable. We have argued that the remedy is to separate two concerns and supply each with the right mechanism. Generality comes from an ontology - an explicit specification of a domain's entities, relationships, and constraints - operated by a single unmodified agent through a small set of generic primitives, so that a new domain is a new specification rather than new code. Feasibility comes from verifiable optimization - a formulate–solve–validate loop over a small taxonomy of classical optimization families, in which no design is committed until it has been checked against the domain's full constraint set, and a failed check drives correction rather than concealment. Together they describe an agent that runs the engineering loop end to end across many physical domains, and that reaches the depth a domain expert demands not by being built for that domain but by reading its ontology. The position we are willing to defend in public is the one we believe will outlast any particular model generation: *for engineering the physical world, represent the domain explicitly, optimize against it, and commit only what you have verified.*

Notation

Symbol	Meaning
$\xi,\ \mathcal{X},\ g,\ c_r$	decision vector, feasible set, objective, constraint r (eq. 1)
$I,\ J$	addresses (demand); candidate equipment sites
$x_{ij},\ y_j$	assign address i to site j; open a facility at j
$\kappa_j,\ f_j,\ c_{ij}$	capacity, install cost of site j; serve-cost of i from j
$d_{ij},\ D$	drop distance; drop-distance limit
$G = (V, E),\ T$	routing graph; terminal set to connect
$z_e,\ f_e,\ c_e$	select edge e; flow on e; cost of e
$\ell_e,\ \rho_{\tau(e)}$	length of e; medium multiplier for e's install type
$a_e,\ w_e,\ \kappa$	cables on e; fibers on e; cable fiber-capacity
$L_i,\ B$	optical loss to address i; optical budget
$\alpha,\ \sigma_k,\ \gamma_c,\ \gamma_s,\ M$	fiber attenuation; splitter loss; connector / splice loss; margin

A The Problem Contract

Optimization instances are passed to the solver layer through a single uniform contract, independent of family and of domain. An instance names its *family* and *problem type*, carries a *model* (the variables, objective, and constraints, populated from the ontology), and may carry *hints* and *spatial requirements*. A result carries a *status* (optimal, feasible, infeasible, timeout, unbounded, or error), an *objective value* and optimality gap where applicable, family-specific result slots (assignments, selected edges, routes, schedules, time series), and *diagnostics* that name the cause of any failure. This uniformity is what lets the same formulated instance be solved by different engines without the agent knowing which, and what makes the validation step's diagnostics consistent across families. The specific engines, the policy that routes an instance to one of them, and the numeric thresholds that govern routing are implementation-specific and are not described here.

B Solver Family Catalog

The body introduces the five optimization families (Table 1) and measures them (Table 4). This appendix documents each with its decision, its canonical form, and the concrete instance we ran in Section 6, so that each family is illustrated by a real example rather than named in the abstract. In every case the agent constructs the instance from an ontology and dispatches it through the single problem contract of Appendix A.

Graph. *Decision:* what connects, and along which path. *Form:* select edges $z_e \in \{0, 1\}$ minimizing $\sum_e c_e z_e$ subject to a connectivity requirement (shortest path, spanning/Steiner tree, flow). *Instance run:* a minimum-cost Steiner tree connecting five placed terminals over a 36-node, 60-edge street grid; solved OPTIMAL at objective 850 m in 3.2 ms (NetworkX), selecting 17 edges through 13 Steiner nodes. *Where it applies:* cable and pipe routing, road-network shortest paths, utility-network tracing and connectivity analysis. *Engine:* NetworkX.

Mixed-integer programming. *Decision:* what to place and select under capacity and budget. *Form:* the capacitated facility-location program of Section 5.3, binary open/assign variables minimizing installed plus connection cost. *Instances run:* FTTH closure placement (40 addresses, 12 candidate sites, capacity 8, 100 m drop limit) - OPTIMAL at US$298,965 in 104.8 ms, 11 closures opened, 39 of 40 addresses served, one correctly left unassigned beyond reach; and water valve placement (30 meters, 9 stations, capacity 6, 150 m service limit) - OPTIMAL at US$142,287 in 40.7 ms, 9 stations, all served. *Engine:* SCIP via OR-Tools.

Vehicle routing. *Decision:* in what order, by which vehicle, under what limits. *Form:* select a tour/route set minimizing travel subject to visiting each location, capacity, and time-window constraints. *Instance run:* a single-vehicle technician tour over a depot and eight service sites; solved OPTIMAL at 10,973 m total

distance in 5.1 s. *Where it applies:* delivery and logistics fleets, field-technician dispatch, inspection and maintenance rounds, waste collection. *Engine:* OR-Tools routing (guided local search).

Constraint programming. *Decision:* when, on which resource, in what sequence. *Form:* assign start times to interval-valued tasks subject to precedence and resource constraints, minimizing makespan. *Instance run:* a five-task construction sequence (trench → duct → pull → splice → test) on a single crew with a precedence chain; solved OPTIMAL at makespan 465 in 120.1 ms. *Where it applies:* construction and deployment crew scheduling, maintenance-window planning, resource and shift assignment, packing. *Engine:* OR-Tools CP-SAT.

Simulation. *Decision:* what happens under the domain's physics. *Form:* propagate state along a network and report steady-state metrics with warnings. *Instance run:* the optical-budget check of Section 5.5 over the path OLT → T3 → closure(1:32) → ONT, accumulating 20.8 dB (1.9 + 18.3 + 0.6) against the 28 dB budget for a 7.2 dB margin; all three links PASS in 1.0 ms. *Where it applies:* optical and RF link budgets, hydraulic pressure and flow in water and gas networks, voltage drop in electric distribution, and discrete-event analysis of field operations. *Engine:* signal-path analyzer (the same family covers hydraulic flow and discrete-event operations).

C Domain Breadth

The worked example treats one domain in depth, but the system's claim is generality. In deployment the same agent and the same primitives operate across ontologies spanning utilities and telecommunications (fiber, water, gas, electric, wireless access), operations and logistics (routing, scheduling, field execution), and several analysis domains. Each is expressed in the four-part form of Section 4.1 - entity types, relationship types, constraints, constants - and bound to one or more skills that target it. Adding a domain is authoring an ontology, not extending the agent; this is the operational meaning of the generality the paper argues for.

D FTTH Ontology Parameters

Table 5 lists representative declared parameters of the FTTH ontology used in Section 5. They illustrate the form of an ontology's constant set - each a named quantity with a unit and a source standard - and ground the claim that the values the agent reasons with are declarations, not code. The values are industry standards (ITU-T G.984, TIA-598); costs are illustrative and region-dependent.

Parameter	**Value**	**Unit**	**Source**
Optical budget (Class B+)	28	dB	ITU-T G.984
Fiber attenuation @1310 nm	0.35	dB/km	ITU-T G.984
Connector-pair loss	0.5	dB	TIA-598
Fusion-splice loss	0.1	dB	TIA-598
Splitter loss 1:8 / 1:16 / 1:32	10.7 / 14.1 / 17.5	dB	ITU-T G.984
Contingency margin	3	dB	practice
Max cascade depth	2	levels	ITU-T G.984
Max drop / distribution / feeder	100 / 150 / 2000	m	practice
Closure capacity (max served)	32	addresses	design rule

Table 5: Representative FTTH ontology parameters. Each is a declared constant the agent reads; replacing the ontology replaces this table with another domain's constants.

Data and Code Availability

The optimization formulations in this paper (capacitated facility location, the Steiner-tree flow model, the path-budget constraint, and the valid inequality of Section 4.5) are standard and fully specified here; the engines that solve them are established, openly documented solvers. The system's implementation - its

dispatch policy, numeric thresholds, formulation-guidance corpus, and infrastructure - is proprietary and is intentionally not released. The ontology parameters of Appendix C derive from public industry standards (ITU-T G.984, TIA-598). Figures and numeric traces are illustrative of system behavior and are not benchmark measurements.

References

[1] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[2] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner Tree Problem.* Annals of Discrete Mathematics, vol. 53, North-Holland, 1992.

[3] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM*, 7(4):326–329, 1960.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] ITU-T Recommendation G.984. *Gigabit-capable Passive Optical Networks (GPON).* International Telecommunication Union.

[6] TIA-598. *Optical Fiber Cable Color Coding.* Telecommunications Industry Association.