

DEAIMER RESEARCH

Termetheus and the Foundations of Environment Based AI Learning

Custom Terminal Tasks for Agent Post Training and Unseen Evaluation

Shahzaib Awan
CEO, Deaimer
7 October 2026

Research perspective | Custom executable datasets

From task instructions to executable experience: how custom terminal environments can support agent learning, domain-specific capability development, and evaluation on unseen problems.

Abstract

An AI agent learns to work through interactions with an environment: it inspects files, interprets tool outputs, changes a system, and observes the consequences. Training data for this form of intelligence must represent more than the final answer. It must provide executable situations in which meaningful actions produce measurable outcomes. Termetheus is Deaimer's offering for custom terminal-style task datasets, combining natural-language objectives with prepared software environments, supporting assets, reference implementations, and outcome checks. This article examines the research value of that format and its practical role in agent development. We connect executable datasets to learning from feedback, sequential decision making, curriculum design, domain transfer, and evaluation on unseen problems. We also explain how buyers can specify useful task distributions and compare candidate collections through a controlled pilot. The central argument is that a dataset's value depends on the capabilities its environments elicit, the diversity of its underlying mechanisms, and the usefulness of its feedback under the buyer's training or evaluation setup. Termetheus supplies a structure for commissioning and organizing such environments. This research perspective presents design analysis and illustrative scenarios, rather than a new model-performance experiment.

Keywords: AI agents; executable datasets; terminal environments; post-training; reinforcement learning; domain transfer; held-out evaluation.

Contents

1 A different unit of AI training data.....	4
2 Why executable environments matter to learning.....	5
3 A terminal task is a compact experimental world.....	6
4 Domain breadth and capability depth.....	8
5 Difficulty as a distribution design problem.....	10
6 How the same environments support different research uses.....	11
7 What makes a dataset useful beyond its size.....	13
8 Designing a custom collection around the buyer.....	15
9 Illustrative custom environment families.....	17
10 Data separation and generalization.....	18
11 A research driven procurement pilot.....	19
12 Quality information at the delivery boundary.....	20
13 The commercial value of executable experience.....	20
14 Conclusion.....	21
References.....	22

1 A different unit of AI training data

A conventional text example is usually a record: an instruction paired with a response, a question paired with an answer, or a document associated with a label. An executable agent task is a situation. It has an initial state, tools, dependencies, constraints, and an objective that the agent must achieve by acting. The resulting data can include a sequence of observations and decisions, the artifacts produced, and an outcome grounded in execution.

This difference becomes important as agents move from describing work to performing it. A model can explain how a database migration should work and still fail to carry out the migration in an unfamiliar project. It can generate a plausible script while overlooking the actual input schema. It can recognize an algorithm in isolation and still choose the wrong representation when resource constraints and existing interfaces matter. Environment-based tasks expose the gap between recognizing a solution and completing a workflow.

The research literature increasingly treats executable settings as a source of training experience. Pan and colleagues combine software tasks with runtime environments and use sampled agent interactions in training. Jain and colleagues investigate the curation of executable software environments at scale. These studies support the importance of environment availability to agent research; their reported model gains should not be interpreted as evidence of a measured effect from Termetheus. [1, 2]

The practical implication is that training infrastructure needs a supply of useful situations. A team may have a strong model, a working agent harness, and sufficient compute, yet still lack a collection of environments aligned with the capabilities it wants to improve. Generic tasks can be useful for early experimentation, but a targeted development program eventually requires control over domains, difficulty, interfaces, and expected behavior.

Termetheus addresses this requirement through custom terminal-style task packages. Its design centers on an agent inspecting a working environment, reasoning about the domain, making changes, and producing an outcome that can be checked. The task is not limited to writing code: a terminal can expose data pipelines, scientific tools, command-line applications, configuration systems, simulations, and domain-specific assets. The environment supplies the concrete context in which the work becomes executable.

A buyer should therefore evaluate the package as research infrastructure. The instruction matters, but so do the dependency graph, initial state, available tools, and the relationship between actions and feedback. An attractive prompt without a usable runtime creates integration work. A working runtime without a meaningful challenge creates activity without much learning value. The useful unit is the complete task and its environment.

1.1 What Termetheus provides

The inspected Termetheus platform describes task packages that include an instruction, a machine-readable manifest, environment build inputs, supporting data, reference-solution assets, and protected outcome-checking components. The manifest records classification, languages, resource assumptions, and execution limits. Versioned submissions and associated evidence support the

management of a collection over time. These are package and platform capabilities, not a claim that every candidate task is already available for licensing.

For a buyer, this structure provides a starting point for a defined dataset delivery. The exact collection can be specified around desired workflows, tools, and capability targets. The delivery agreement should identify which components are included, which are withheld from the solving agent, and which are licensed for training, internal evaluation, or both. Dataset scope should be settled at the task-package level rather than inferred from a general product description.

Termetheus and ColdStart serve related but distinct purposes. ColdStart organizes an inquiry into reliable recovery of broken systems. Termetheus covers a broader supply of terminal-accessible problems and custom environments for task completion, agent development, and evaluation. A Termetheus collection can include debugging or recovery scenarios, but its research value also extends to construction, transformation, analysis, and domain-specific problem solving.

2 Why executable environments matter to learning

Language supervision teaches a model which responses are likely to be appropriate. Execution feedback adds information about whether an intervention actually worked in a particular environment. That information is valuable when surface plausibility and functional correctness diverge. A program may read well but fail on a boundary case; a configuration may look reasonable but conflict with the installed dependency; a report may be fluent but inconsistent with the provided data.

Le and colleagues explore reinforcement learning for code generation using feedback associated with functional correctness. Yao and colleagues study the interleaving of reasoning and actions, where actions obtain observations that can inform subsequent decisions. The connection to Termetheus is conceptual: an executable task can generate observations that a static answer record does not contain. Whether those observations improve a buyer's model depends on the training method, task distribution, and quality of collected experience. [3, 4]

2.1 Observation and action form a learning loop

An agent typically begins with incomplete knowledge of its environment. It needs to locate relevant files, discover tool behavior, inspect a schema, or reproduce an observed failure. Each observation changes what it can reasonably do next. The task therefore creates a loop in which information gathering and intervention are connected.

For example, a data-processing task may require the agent to inspect multiple files before discovering that one source uses a different identifier convention. Writing a transformation before examining that difference may produce an output that passes a superficial format check but fails the actual reconciliation objective. The experience can teach a strategy: establish representation and constraints before committing to a transformation.

This loop produces a richer record than a final script alone. The script shows the implementation, while the interaction shows how the agent found the necessary facts and responded to evidence.

For some learning objectives, successful final artifacts are enough. For others, trajectories help distinguish an effective strategy from a fortunate result.

2.2 Feedback should match the capability

A feedback signal is useful when it distinguishes behavior relevant to the target skill. If the objective is complete data reconciliation, checking only file existence supplies a weak signal. If the objective is efficient numerical computation, checking only one tiny input may ignore the intended resource constraint. The buyer needs a task whose observable outcomes express the capability it wants to train.

The same principle applies to feedback granularity. A terminal reward may provide a clear completion signal but little information about intermediate mistakes. Additional task-specific diagnostics can help a training pipeline identify partial progress or recurring failure patterns. Those diagnostics should remain aligned with the true objective and should be included only where the agreed package and runtime support them.

More feedback is not always better. Excessively revealing diagnostics can turn the task into trial-and-error reconstruction of a hidden answer. A collection should distinguish feedback suitable for training from evidence reserved for held-out evaluation. The desired visibility is a design choice, not a property that should be left to accidental file placement.

2.3 The environment controls what experience is possible

The environment determines which actions are available and which consequences the agent can observe. A task containing only one isolated function elicits different behavior from a task containing a service, a data store, and a client that depends on them. Both can be useful, but they support different questions about agent capability.

An environment can be made complex without becoming useful. Numerous files, long logs, or elaborate dependencies may increase search effort while adding little conceptual depth. A better design introduces interactions that matter: a representation mismatch, a concurrency condition, a resource limit, or an interface that must remain compatible. These conditions require the agent to reason about the system rather than merely process more material.

The goal is task-relevant richness. A custom environment should contain enough context to require the intended skill and enough control to make its outcomes interpretable. That balance gives buyers a way to commission environments that test their actual development hypotheses.

3 A terminal task is a compact experimental world

The terminal is a flexible access surface. It can expose a compiler, an interpreter, a database client, a simulator, or a command-line utility through a common interaction channel. This makes terminal-style tasks attractive for combining multiple technical domains without requiring an entirely different interface for each one.

The agent still needs domain knowledge. A common shell does not make a hardware simulation equivalent to a database problem. Instead, it allows heterogeneous workflows to be packaged into a

consistent execution setting. The learning challenge lies in selecting and using the domain tools correctly within that setting.

3.1 Instructions define the objective

A task instruction should tell the agent what outcome to achieve and which constraints matter. It should avoid dictating every intermediate step unless following a specified procedure is itself the target capability. An outcome-oriented instruction permits different valid approaches and makes it possible to observe the agent's planning choices.

This is valuable for custom datasets because buyers often care about generalizable competence. If every task names the exact command sequence, the collection may measure compliance with a recipe. If the objective is clear while the implementation is open, the environment can elicit discovery, planning, debugging, and adaptation.

3.2 Initial state makes the problem concrete

The initial state establishes what the agent must work with. It may include a partially implemented repository, malformed inputs, incompatible dependencies, a prepared simulation, or a service configuration. That state converts an abstract request into an executable problem.

The distinction matters for realism. Asking an agent to invent a pipeline from nothing tests a different capability from asking it to integrate with an existing pipeline and preserve its interfaces. Asking it to calculate a result from an embedded example differs from asking it to inspect a collection of files and choose the right analysis. Custom environments can encode these differences directly.

3.3 Dependencies carry part of the challenge

Dependencies should be selected for their role in the workflow. A compiler version can affect language features; a library interface can affect the repair; a simulator can determine which design behaviors are observable. The task package should identify the execution assumptions needed to reproduce the intended work.

Stable dependencies help a buyer interpret experiments. If an environment changes between training runs, a shift in outcomes may reflect dependency drift rather than a new policy. Conversely, deliberate dependency variation can be useful when the goal is robustness across versions. The important distinction is whether variation is controlled and recorded.

3.4 Artifacts make outcomes inspectable

An agent task commonly produces files, code changes, structured outputs, or executable behavior. These artifacts connect the agent's actions to a result that can be inspected independently of its final explanation. A report, for example, should be evaluated against the provided inputs rather than accepted because the agent claims to have completed it.

Artifact design also affects integration. A buyer may need predictable paths, schemas, and conventions to collect training records or run batch evaluations. Agreeing these interfaces in

advance reduces the work required to incorporate a dataset into an existing pipeline. Packaging is therefore part of the dataset's utility, not merely an administrative step.

3.5 Reference assets support development

Reference implementations can help buyers understand an admissible solution and diagnose environment issues. They may also support demonstration generation or supervised learning when licensed for that purpose. Their visibility must be separated from the agent-facing environment in evaluation settings.

A reference implementation should not be confused with the only valid strategy. Where the objective permits alternatives, the useful concept is satisfaction of the task contract. A model that reaches the required outcome through a different sound approach can demonstrate precisely the flexibility a buyer wants to develop.

4 Domain breadth and capability depth

A collection becomes more useful when domain coverage and capability coverage are considered together. Many tasks in different-looking folders can still require the same strategy. Conversely, several tasks in one domain can demand substantially different forms of reasoning. A buyer needs to know what varies beneath the surface.

The Termetheus taxonomy supports software, machine learning, science, operations, security, hardware, and media categories. These are organizational categories in the platform. The breadth and volume of an actual deliverable should be established through its catalog and agreed scope. A supported category is not evidence that a ready-to-license inventory exists in every subfield.

Table 1 Domain scope and illustrative environment choices

Domain	Capability target	Environment elements
Software and data	Repair, construction, transformation	Repositories, services, schemas, local datasets
Machine learning	Pipeline reasoning and computational tradeoffs	Model code, fixtures, configuration, metrics
Science	Numerical and domain reasoning	Scientific libraries, input data, simulations
Operations	Rule-based analysis and reconciliation	Records, business constraints, output contracts
Security	Authorized analysis and systems reasoning	Isolated artifacts, local services, controlled inputs
Hardware	Engineering tool use and interpretation	Design files, toolchains, simulation inputs
Media	Structured asset processing	Local assets, transformation tools, format requirements

4.1 Software and data engineering

Software environments can require the agent to understand existing interfaces, repair cross-file behavior, implement an algorithm, reconcile structured inputs, or adapt a dependency. The task's value depends on the reason the problem is difficult. A cross-component invariant creates a different challenge from an isolated syntax error.

Data engineering is particularly suitable for environment-based tasks because errors can be hidden behind well-formed outputs. An agent may need to infer schema relationships, handle missing values, preserve identifiers, and verify completeness. The environment can expose both the raw assets and the tools needed to transform them, making representation choices part of the work.

4.2 Machine learning workflows

Machine-learning tasks can involve preparing inputs, fixing an inference pipeline, interpreting evaluation outputs, or optimizing a computation. Some useful problems can be designed around modest local resources, while others genuinely require accelerators or external infrastructure. The current inspected baseline collection policy is a constrained Linux setting; accelerator-backed work needs a separately agreed runtime.

The research value lies in the agent's coordination of technical steps. It may need to inspect a feature definition, identify an incorrect preprocessing order, and produce a consistent evaluation artifact. The environment lets the buyer test whether the agent connects those steps correctly instead of simply explaining each concept in isolation.

4.3 Scientific computing

Scientific tasks can make numerical assumptions and domain conventions observable. An agent may need to choose an appropriate transformation, reconcile units, implement a calculation, or interpret a simulation output. The task should make relevant data and tools available while clearly defining the expected result and tolerances.

Scientific authenticity requires more than scientific terminology. A problem should depend on the discipline's actual reasoning or computational structure. Domain-specific review and carefully selected assets can help buyers commission environments that are useful to their intended research applications.

4.4 Security and systems

Controlled security tasks can involve artifact analysis, defensive diagnosis, or correction of an access-control defect. The environment should keep the activity within an authorized and reproducible scope. The expected capability should be explicit: understanding a protocol, tracing a failure, or restoring a required trust property.

Systems tasks can test reasoning about resources, processes, files, and configuration. Their difficulty often comes from interactions among components and from the need to distinguish

symptoms from causes. The terminal provides an interface to those interactions, but task design determines whether the agent is required to understand them.

4.5 Hardware and engineering tools

Hardware-oriented terminal environments can expose simulation, design artifacts, and command-line engineering workflows. A candidate task may require interpreting a failed simulation, correcting a design property, or producing an artifact consistent with specified constraints. These are examples of potential custom scope rather than disclosures of the private task pool.

Such environments broaden the meaning of a coding-agent dataset. The agent may write source code, but correctness depends on a domain model or engineering requirement. For buyers targeting specialized technical assistance, this can make the collection more relevant than a large set of general programming exercises.

5 Difficulty as a distribution design problem

A buyer does not always need the hardest possible task. Training and evaluation have different relationships to difficulty. A development curriculum may need accessible tasks that produce successful experience, intermediate tasks that expose recoverable mistakes, and demanding tasks that reveal remaining limitations. A private holdout may emphasize challenging mechanisms that distinguish strong systems.

Difficulty is also relative to the evaluated configuration. A task that challenges one model may be straightforward for another model using a better tool interface. Yang and colleagues show that the agent-computer interface can materially affect software-task performance. Dataset specifications should therefore describe the intended model-agent setup or at least the relevant observation and action interfaces. [5]

5.1 Separate complexity from inconvenience

Intrinsic complexity arises when a solution requires reasoning about constraints, dependencies, or domain structure. Inconvenience arises from irrelevant search, ambiguous instructions, broken setup, or arbitrary formatting traps. Both can reduce pass rates, but only the first reliably supports the buyer's capability objective.

A useful custom collection should make the central challenge explainable. The task can require substantial work without being obscure. When an agent fails, the failure should tell the buyer something about its behavior, such as a representation mistake or an incomplete plan, rather than simply reveal that the environment was inaccessible.

5.2 Vary one mechanism deliberately

Controlled variants can support targeted learning. A family of tasks might vary the input distribution, dependency interaction, or required algorithm while retaining a common workflow. This gives the training pipeline a set of experiences related to one capability.

Cosmetic variation is less informative. Renaming files or changing constants while preserving the exact solution structure can increase task count without adding much independent coverage. A buyer should distinguish the number of instances from the number of underlying mechanisms and task families.

5.3 Compose mechanisms to extend the horizon

Longer tasks can combine several dependent stages. A solver may need to prepare data before reproducing a failure, repair the implementation before running a transformation, and inspect the result before deciding it is complete. The dependency between stages gives planning a meaningful role.

Simply adding more commands does not necessarily extend the reasoning horizon. The relevant question is whether later actions depend on earlier discoveries and whether mistakes propagate through the workflow. A thoughtfully composed task can expose this structure without becoming large or expensive to run.

5.4 Match difficulty to the feedback budget

Very difficult tasks can provide little positive training experience if the current policy rarely succeeds and the reward is sparse. A buyer may address this through demonstrations, a staged curriculum, additional feedback, or a stronger initial policy. Which approach is appropriate depends on the training experiment.

The dataset supplier can help by designing task families at different levels of complexity. The buyer should retain control of how these families are sampled and how exposure changes during training. A useful curriculum is an experimental policy over tasks, not a label attached to a collection once and forgotten.

6 How the same environments support different research uses

An executable task can be used in several ways, but those uses require different data visibility and split policies. A training set can expose reference assets to a demonstration generator. An evaluation set must preserve its unseen status. A diagnostic collection may prioritize interpretable failure cases over broad score comparison.

Table 2 Research uses of executable task packages

Use	Useful data	Design requirement
Supervised learning	Successful trajectories and artifacts	Record sufficient environment context
Reinforcement learning	Episodes, observations, outcome feedback	Reset environments and define feedback visibility
Preference learning	Comparable attempts and documented outcomes	Compare behavior under equivalent conditions

Use	Useful data	Design requirement
Unseen evaluation	Held-out environments and result summaries	Separate task families and track exposure
Harness research	Tool interactions and execution records	Control model and task conditions

6.1 Supervised trajectory learning

An environment can support the generation or collection of successful action sequences. These trajectories may be used for supervised fine-tuning when the licensing scope and data format permit it. The record can include commands, observations, modifications, and a verified outcome rather than only the final answer.

Successful trajectories need selection. A run may reach the objective through unnecessary detours or fragile assumptions. A training pipeline should consider whether the behavior represents the strategy it wants to teach. The reference solution can help interpret the task, but it should not force every training trajectory into one exact implementation.

6.2 Reinforcement learning with executable feedback

A resettable environment and an outcome signal can support reinforcement learning experiments. The agent interacts with a sampled task, receives observations, and is updated according to the buyer's chosen algorithm. The task package supplies the setting; it does not itself supply a complete model-training system.

At a conceptual level, the buyer chooses a distribution over task environments and optimizes a policy for expected return under that distribution. The definition of reward and the representativeness of the task distribution jointly determine what behavior the training process encourages.

$$J(\theta) = \mathbb{E}_{e \sim D, \tau \sim \pi_{\theta}(\cdot | e)}[R(\tau, e)]$$

In this expression, θ represents the policy parameters, D is the chosen task distribution, τ is an interaction trajectory generated in a sampled environment, and R is the specified return. This is a general learning formulation, not a proprietary Termetheus algorithm. It makes the buyer's design responsibility explicit: an improvement on one distribution does not automatically imply general competence on every workflow.

A terminal completion signal can be combined with other training signals if the pipeline supports them. Any additional shaping should be examined for unintended incentives. A reward for short trajectories, for instance, can discourage useful inspection unless it is subordinate to correct task completion.

6.3 Preference and comparative behavior data

Multiple attempts on the same task can produce candidate behaviors for comparison. A buyer may prefer a trajectory that completes the objective with less unnecessary intervention, better use of evidence, or lower cost. Such preferences can support a separate learning process when the criteria are observable and consistently applied.

A preference label should not disguise a hidden correctness requirement. If one attempt fails the task, that outcome belongs in the record. If two attempts both succeed, the comparison can concern style of execution or resource use. Keeping these distinctions clear makes the data more useful for targeted training.

6.4 Evaluation on unseen environments

A private task collection can assess systems on incidents and workflows they have not encountered during development. The evaluation must identify the tested model-agent configuration, task versions, and execution envelope. The result measures performance under those conditions.

Repeated attempts can expose variability. A system that succeeds once may not be reliable across repetitions, and a pooled score can hide failure on an entire domain. Buyers should retain per-task outcomes and enough metadata to analyze where performance is concentrated. These are general reporting principles, not disclosures of internal review procedures.

6.5 Harness experimentation

The same environments can support experiments on context management, tool interfaces, execution policies, and agent planning. Holding the task panel fixed while changing the harness makes it easier to observe which behaviors change. A buyer may learn that a model's weakness is partly an interface problem rather than a lack of domain knowledge.

A native-system comparison and a controlled-harness comparison answer different questions. The former asks which complete system performs better; the latter supports more specific attribution to a changed component. The dataset can support both if the experimental record preserves the distinction.

7 What makes a dataset useful beyond its size

A large task count is a limited description of a collection. Two datasets with the same number of packages may differ in mechanism diversity, runtime reliability, domain depth, and ease of integration. A buyer should assess the amount of relevant experience the collection can produce within its actual compute and operational budget.

Kapoor and colleagues argue that agent evaluation needs attention to cost, holdouts, and reproducibility alongside accuracy. That perspective is relevant to dataset procurement: more examples can be less useful if they are redundant, expensive to initialize, or too closely related to the intended test set. [6]

7.1 Effective diversity

Effective diversity concerns the independence of the challenges represented. Several database tasks can test different capabilities: schema interpretation, concurrency, query optimization, or migration compatibility. Several tasks from different domains can nevertheless depend on the same simple text transformation.

The catalog should therefore expose task families and primary capability targets where possible. Buyers can then select a mixture aligned with their research hypothesis. This is more informative than relying solely on category names or language counts.

7.2 Executability

An executable dataset must run in the buyer's setting. Environment build assumptions, architecture, resource requirements, network behavior, and runtime interfaces all affect this property. A task that works in a supplier's development environment may still need adaptation before it works in a buyer's training loop.

A controlled pilot should test this integration directly. The buyer can sample representative packages, initialize them in its runtime, collect outcomes, and check whether the resulting records fit its data pipeline. Integration success is an observable property that should precede claims about scale.

7.3 Reusability

A resettable task can generate multiple experiences from different policies or repeated attempts. This makes its research value different from a single static answer record. The same environment can help study progress over model versions or the consequences of a changed tool interface.

Reusability has limits. Frequent exposure can make an evaluation task familiar. A training environment can also become exhausted as the policy learns its mechanism. The buyer should distinguish reusable training worlds from protected holdouts and plan collection renewal accordingly.

7.4 Operational efficiency

Training value depends partly on the cost of obtaining experience. Initialization time, runtime, failed builds, storage use, and record processing all consume resources. A dataset with a lower price per package may still be expensive if its environments are difficult to operate.

A buyer can track the rate of completed usable trajectories per unit of compute or cost, alongside task outcomes. This is a procurement and experiment-management measure, not an official Termetheus score. It helps connect the purchase to the actual training budget.

7.5 Documentation and provenance

Documentation should describe the task's purpose, execution assumptions, expected artifacts, and permitted uses. Provenance should identify the basis on which task assets can be licensed and

distinguish authored material from third-party dependencies. The exact rights granted belong in the delivery agreement.

Private or unpublished material is not automatically exclusive. A buyer seeking exclusivity needs its scope defined: which packages, what uses, which period, and which transfer or sublicensing rights. Precise terms make the dataset easier to use confidently without turning the article into a legal claim.

8 Designing a custom collection around the buyer

The strongest custom dataset begins with a capability specification. A request for difficult terminal tasks is a starting point, but it leaves important design choices open. A request to improve multi-step data engineering under constrained resources is more actionable because it identifies the workflow and the capabilities to emphasize.

The specification should connect each desired capability to a class of executable situation. This makes it possible to choose assets and tools for a reason and to assess whether the delivered mixture matches the research objective.

8.1 Define the target workflow

Buyers should identify what their agent is expected to do. Examples include modifying existing software, integrating data sources, carrying out numerical analysis, diagnosing a system, or using a specialized engineering tool. These workflows differ in their knowledge requirements and their relationships to state.

The specification can also identify the intended user setting. An agent for expert developers may receive a concise technical instruction. An agent for a general business workflow may need more contextual description. Custom tasks should reflect the observation and instruction conditions relevant to the buyer.

8.2 Define the capability mixture

A collection can combine tasks targeting discovery, planning, implementation, debugging, and verification of outcomes. The proportions should follow the development hypothesis. If the buyer's model already handles local implementation well but struggles with representation choices, a larger share of diagnostic and data-interpretation tasks may be appropriate.

Coverage can be recorded in a matrix rather than inferred from prose. The matrix helps identify underrepresented skills and prevents the collection from accumulating many tasks that test the same convenient mechanism.

8.3 Define the runtime envelope

The buyer should specify its execution environment, available tools, resource limits, and network policy. The supplier can then design packages that fit those assumptions. If an incident requires a tool or resource outside that envelope, it should be identified early.

Runtime compatibility is particularly important when the dataset will be sampled frequently during training. Small setup differences multiplied across many attempts become substantial operational work. A compatible package format and stable output interface make the collection easier to incorporate into existing infrastructure.

8.4 Define the data products

A task package and a trajectory dataset are related but different deliverables. The package allows execution; trajectories are observations collected from executions. Reference solutions, model-generated attempts, outcome records, and analytical reports should be listed separately when they are included.

A buyer should also specify whether it needs only training environments, an independent evaluation pool, or both. If both are requested, their separation should be maintained in the catalog and delivery process. The size of each pool should reflect its use rather than an arbitrary split percentage.

8.5 Define commercial scope

Commercial scope can include an existing agreed collection, newly authored environments, adaptation to a buyer's runtime, or a combination. Task domains, deliverable counts, rights, and support should be explicit. This article does not set a universal price or imply that all possible custom configurations are immediately available.

The research advantage of customization is alignment. A buyer can invest in experiences that test a known weakness or support a planned experiment, rather than purchase a broad collection and hope that a useful subset exists within it.

Table 3 A buyer specification for a custom collection

Scope decision	What to specify
Capability goals	Workflows and decisions the agent should learn
Domain coverage	Tools, languages, assets, and domain assumptions
Runtime	Compute, dependencies, network access, and reset behavior
Data products	Packages, trajectories, reference assets, metadata, and evidence
Difficulty mixture	Mechanisms, interaction depth, and intended curriculum
Evaluation split	Held-out families and rules governing access
Commercial scope	Permitted uses, rights, adaptation, and delivery versions

9 Illustrative custom environment families

The following examples are constructed design scenarios. They are not protected Termetheus task descriptions or claims about the current licensed inventory. They show how a buyer's capability goal can be expressed through a terminal-accessible environment.

9.1 Reconciling heterogeneous data

An agent receives several structured sources with inconsistent identifiers, missing values, and different representations of the same entities. It must produce a coherent output that preserves the required relationships and accounts for all relevant records. The environment supplies representative inputs and tools for inspection and transformation.

The learning target is representation reasoning. A successful strategy requires understanding the data before committing to a mapping, handling ambiguity within the stated rules, and checking that the final artifact is complete. Surface variation can be introduced through new schemas and distributions, while mechanism variation can involve different reconciliation structures.

9.2 Repairing a cross component software defect

A repository contains a failure whose cause spans an interface between components. The agent must reproduce the behavior, inspect the relevant interaction, make a scoped change, and demonstrate the expected output. The problem is designed so that a local patch to one observed case does not resolve the broader contract.

The learning target is coordinated reasoning across code and execution feedback. The environment creates opportunities for the agent to update its hypothesis after a test or tool output contradicts its initial assumption. A family of such tasks can vary the interface mechanism rather than only the repository's appearance.

9.3 Choosing a numerical method under constraints

An agent must compute or transform an artifact subject to a documented accuracy requirement and a limited resource envelope. A simple method may be correct on small inputs but unsuitable for the provided scale. The environment makes the input structure and resource conditions inspectable.

The learning target is the relationship between mathematical correctness and computational feasibility. The task can reward choosing an appropriate method, checking numerical assumptions, and producing a reproducible output. Its usefulness depends on whether the constraints represent the buyer's actual applications.

9.4 Using an engineering simulator

An agent receives design artifacts and a simulator-backed objective. It must understand the relevant requirement, inspect failure evidence, and modify the design or supporting inputs so that the specified behavior holds. The command-line simulator supplies feedback tied to domain semantics.

The learning target is technical reasoning grounded in a model of the engineered system. Code generation is one part of the work, but a syntactically valid artifact is insufficient. This family can support specialized agents whose value depends on interpreting engineering constraints.

9.5 Producing a complete analytical artifact

An agent must build a report or structured output from supplied assets, using a defined schema and transformation rule. Some inputs require filtering, joining, or normalization. The environment makes completeness and consistency observable through the relationship between source records and the final artifact.

The learning target is end-to-end workflow completion. A plausible explanation of the result is not enough; the artifact must reflect the underlying data. This type of task can be useful for agents operating in analysis and operations settings as well as for software-development agents.

10 Data separation and generalization

A custom environment dataset is most useful when it supports learning without weakening the evidence used to judge that learning. Training and evaluation should therefore be treated as separate products or clearly separated partitions. The separation needs to account for task mechanisms and assets, not only file names.

10.1 Hold out mechanisms and families

Closely related variants can leak solution structure across a split. A model trained on several examples of one mechanism may succeed on a renamed version without demonstrating broader generalization. Holding out families or selected mechanisms creates a more demanding test of transfer.

This does not mean related variants are undesirable. They can be useful during training. The issue is the claim made by the evaluation. A within-family test measures a different kind of generalization from a cross-family test, and the report should identify which one was performed.

10.2 Track exposure

A task can become familiar through training, prompt development, repeated evaluation, or publication. Exposure tracking helps the buyer preserve the meaning of an unseen result. If a private evaluation package becomes a public sample, its role should be reconsidered.

Dataset renewal can address accumulated exposure. A supplier can commission new families or replace known assets while preserving the capability target. Meaningful renewal changes the reasoning problem rather than merely obscuring a familiar solution with new names.

10.3 Interpret transfer carefully

An improvement on one task distribution is evidence about that distribution. Claims about other domains require separate evaluation. A stronger policy on database tasks may transfer to some data-processing workflows, but that relationship should be tested rather than assumed.

The research value of custom datasets includes making transfer questions explicit. A buyer can design a training mixture and a separate evaluation mixture to test whether experience with one class of environments improves behavior elsewhere.

11 A research driven procurement pilot

A pilot should answer a concrete question about the dataset's usefulness. It can test whether packages integrate with the buyer's runtime, whether task mechanisms match the requested capabilities, and whether the resulting experience supports the planned experiment. These questions are more actionable than an impression that the tasks look sophisticated.

11.1 Inspect representative packages

The pilot should include representative task families and resource profiles. Buyers can inspect the instruction, environment assumptions, output contract, documentation, and rights information. They should confirm that the assets included in the delivery match the intended training or evaluation use.

The supplier should distinguish development fixtures from substantive task samples. A deliberately simple starter package can demonstrate compatibility, but it cannot establish domain depth or the difficulty of the licensed collection.

11.2 Test integration before scale

Representative tasks should run through the buyer's intended execution loop. The pilot can record initialization behavior, runtime, output collection, and failed integrations. These observations help estimate the operational cost of a larger delivery.

A task collection that requires substantial adaptation may still be useful, but that work should be included in the plan. Discovering integration requirements early is better than treating them as an unexplained training failure later.

11.3 Measure a development hypothesis

If the buyer intends to train a model, the pilot can compare a baseline with a controlled training condition using the candidate environments. The evaluation should use separate held-out tasks and preserve the model-agent configuration and budget. The report should distinguish changes in model parameters from changes in the harness.

If training is outside the pilot scope, the buyer can still examine the distribution of outcomes and failure mechanisms from its current agent. This provides evidence about whether the tasks expose relevant limitations and whether the feedback is interpretable.

11.4 Consider more than one endpoint

Useful pilot measures include integration success, task outcomes, capability coverage, experience collection cost, and generalization on a held-out pool. The primary measures should be chosen

before observing results. Exploratory findings can guide a later collection, but they should not silently replace the pilot’s original objective.

Table 4 Questions for a controlled procurement pilot

Question	Observable evidence
Does it integrate?	Successful builds, resets, and harness execution
Does it target the right skills?	Review of task mechanisms and sampled agent behavior
What does experience cost?	Build time, episode time, storage, and compute use
Are outcomes interpretable?	Readable artifacts, documented constraints, useful diagnostics
Does it support the learning hypothesis?	Controlled comparison on a separate evaluation collection

The pilot should end with a decision about scope: which task families are useful, which need adaptation, what runtime changes are required, and what evidence supports proceeding to a larger collection. This connects the purchase to an actual research program rather than to a nominal count of files.

12 Quality information at the delivery boundary

Buyers need confidence that a task package is usable and that its stated outcome has meaningful support. Termetheus organizes versioned submissions and review evidence around the package. The buyer-facing delivery should state the status of each included task and provide the agreed evidence summary, without treating incomplete checks as completed results.

Detailed internal review procedures are outside the scope of this article. The relevant commercial principle is that status should be attached to the exact delivered version. Runtime requirements, known limitations, and available execution evidence should be visible enough for the buyer to assess readiness. Model-performance claims should identify the configuration and trials that support them.

13 The commercial value of executable experience

Termetheus is designed for teams that need terminal-style task datasets and custom environments for agent research. The offer is concrete: specify the capabilities, domains, runtime, and intended data use, then organize a collection of executable problems around those requirements. A defined collection can support post-training, controlled evaluation, and analysis of agent behavior.

Its value is strongest where a buyer’s development target is specific. A team working on scientific computing may need domain-authentic assets and numerical constraints. A team improving software agents may need cross-component defects and meaningful repository context. A team building analytical assistants may need data reconciliation and complete artifact production. Custom environments allow those requirements to shape the data directly.

The supplier and buyer have complementary roles. Deaimer can design and package the environments within an agreed scope. The buyer chooses the model-training method, experimental setup, and deployment interpretation. A dataset can create useful learning opportunities, but it does not independently guarantee a model improvement or deployment outcome.

A clear licensing and delivery agreement turns those opportunities into an actionable product. The agreement should identify package versions, included assets, permitted uses, and any exclusivity or adaptation requirements. A pilot can then demonstrate whether the collection fits the buyer's infrastructure and research goals before the scope expands.

For teams seeking a tailored collection, the starting point is a short capability brief: what the agent should learn to do, which domains matter, and which runtime it will use. Deaimer can use that brief to define an appropriate sample and proposed dataset scope. Technical and commercial discussions can begin through www.deaimer.com.

14 Conclusion

Executable agent datasets provide situations in which models can discover information, take actions, observe consequences, and produce inspectable outcomes. Their research value comes from the structure of the experience: the environment, the task contract, the feedback, and the distribution of mechanisms across the collection.

Termetheus brings this format to custom terminal-style tasks. Its package structure supports domain-specific environments and documented execution assumptions, while its broader platform organizes versioned work. For buyers, the important question is how an agreed collection fits a training or evaluation program and what capabilities it can meaningfully expose.

The strongest dataset investment begins with a precise capability goal and ends with evidence from a controlled pilot. Task count, domain breadth, and apparent complexity are useful descriptors, but they do not replace executability, effective diversity, and alignment with the buyer's experiment. Termetheus's commercial proposition is to make that alignment concrete through commissioned environments and clearly scoped dataset deliveries.

References

- [1] Pan, J., Wang, X., Neubig, G., Jaitly, N., Ji, H., Suhr, A., and Zhang, Y. Training Software Engineering Agents and Verifiers with SWE-Gym. ICML 2025. arXiv:2412.21139. <https://arxiv.org/abs/2412.21139>
- [2] Jain, N., Singh, J., Shetty, M., Zheng, L., Sen, K., and Stoica, I. R2E-Gym: Procedural Environments and Hybrid Verifiers for Scaling Open-Weights SWE Agents. 2025. arXiv:2504.07164. <https://arxiv.org/abs/2504.07164>
- [3] Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. NeurIPS 2022. arXiv:2207.01780. <https://arxiv.org/abs/2207.01780>
- [4] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing Reasoning and Acting in Language Models. ICLR 2023. arXiv:2210.03629. <https://arxiv.org/abs/2210.03629>
- [5] Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. NeurIPS 2024. arXiv:2405.15793. <https://arxiv.org/abs/2405.15793>
- [6] Kapoor, S., Stroebl, B., Siegel, Z. S., Nadgir, N., and Narayanan, A. AI Agents That Matter. 2024. arXiv:2407.01502. <https://arxiv.org/abs/2407.01502>

Source note Termetheus package and platform statements were checked against Deaimer’s platform documentation and source snapshot on 7 October 2026. Academic works are cited for research context; the article does not claim their methods as Deaimer inventions or their reported results as Termetheus results. Illustrative task families are design examples, and specific inventory, rights, availability, and delivery terms depend on the agreed collection.