ChatGPT

Enhancing the PulZ AI Agent System: 2026 State-of-the-Art Update

Recap of the Current Instruction Pack & Capabilities

Your **PulZ** system blueprint (the "instruction pack") already lays a **strong foundation**. It proposes an **autonomous AI orchestration agent** with capabilities that go beyond existing tools like Zapier, LangChain, AutoGPT, etc. Key features already covered include:

- **Natural Language Goal Parsing & Planning:** Users can give free-form goals; the system parses these into structured objectives and generates multi-step execution plans autonomously [1] [2]. It even considers multiple strategy options with cost/success estimates (e.g. the plan outlines generating 3-5 possible approaches) [3]. This addresses the gap of **autonomous workflow generation** that current systems lack [4].

- **Dynamic Tool Discovery and Use:** Rather than relying only on pre-built integrations, PulZ can **auto-discover available APIs/tools**, understand their interfaces, and compose them as needed in real-time [5]. This "**Dynamic Tool Ecosystem**" means if a new API or a robotic peripheral is available, the agent can integrate it on the fly (reading schemas, self-documenting JSON specs, etc.) [6]. This is cutting-edge compared to static node-based flows in RPA platforms.

- **Memory & Learning Across Sessions:** The design incorporates a memory module (vector stores, working memory, episodic store) to maintain context and **learn from past executions** [7]. Patterns of successful strategies are retained in a **Pattern Library** for reuse [8]. This cross-session learning (inspired by ideas like Reflexion and meta-learning) gives the agent continuous improvement, a feature missing in typical single-run agents.

- **Multi-Layer Safety and Resource Management:** PulZ's plan puts heavy emphasis on safety and control. It has a **Budget Guardian** to enforce token/cost limits, a **Validation Engine** for result correctness, a **Loop Detector** to catch infinite loops, and a **Human-in-the-Loop** gate for high-risk actions [9]. Additionally, a **Security Sandbox** and audit logging are included for safe execution and transparency [10] [11]. These safeguards address hallucination, runaway costs, and other risks that early AI agents struggled with. Indeed, the system explicitly aims to prevent known failure modes (hallucinations, infinite loops, unauthorized actions) which are **pain points in current AI agent frameworks** [12] [13].

- **End-to-End Autonomy with Oversight:** The envisioned execution flow – from goal input to strategy, tool use, real-time monitoring, validation, and learning – is comprehensive [3] [14]. An example in the docs shows the agent handling a complex research task, autonomously choosing parallel web scraping, validating data, and producing a final report under budget [15] [16]. This demonstrates the intended **"self-orchestrating"** ability of PulZ to handle complex workflows with minimal human intervention (aside from configured approval checkpoints).

Bottom Line: The instruction pack as it stands is quite **thorough and forward-thinking**. It combines elements that no single existing system had, as you noted [4]. In 2024-2025, many AI devs were still wiring together LangChain or AutoGPT loops, often hitting issues with memory or safety. PulZ's design anticipated those issues and baked in solutions (persistent memory, validations, etc.). So yes – the core **objectives and architecture are solid and "already there"** in your documents.

The question now is: what **additional** considerations or cutting-edge updates (as of **2026**) could strengthen PulZ even more? Below, we delve into new developments and high-end features that you might integrate into the system moving forward.

Integration of Physical World & Multi-Modal Capabilities

One area to extend PulZ is enabling it to operate **beyond purely digital tasks** – for example, interfacing with robotics, IoT devices, or other physical systems. Your use cases mentioned **running a 3D printing company** or **unloading a shipping container with a robotic arm**. To support these, the PulZ architecture should incorporate **multi-modal inputs** and **physical action interfaces**:

- **Robotics as Tools/Peripherals:** Treat hardware controllers (robot arms, drones, printers) as just another class of "tools" the agent can utilize. In practice, this means providing an API or command interface for each device. For example, a robotic arm might expose actions like `move_arm(x,y,z)`, `grip()`, with safety constraints (payload limits, etc.). PulZ's tool schema can describe these capabilities so the agent knows how to call them [17]. The agent becomes a high-level manager issuing commands to the robot's low-level controller (which is already programmed for basic motions). This **modular tool interface** approach is flexible – whether it's a web API or a local ROS (Robot Operating System) action server, PulZ can integrate it by reading its spec and constraints.

- **Sensor and Perception Input:** Physical tasks require perceiving the environment. High-end solutions as of 2025-2026 use **multimodal AI** to combine vision, language, and sensor data. For PulZ, you can incorporate a vision model or feed camera/LiDAR inputs into the agent's context. Notably, Google's **PaLM-E** (Embodied multimodal model) demonstrated that a single large model can process visual observations and text to guide robots [18]. While you might not build a giant multimodal model from scratch, you can use pretrained ones or specialized CV models. For instance, the agent could call an object recognition tool on camera feed to get a structured description of a scene ("10 boxes detected, stacked 2 high"), then plan robot actions accordingly. **Force feedback** ("feels") from a robotic gripper can be treated similarly – e.g. if sensors indicate a slip, the agent could decide to readjust grip or slow down. The key addition is a **Perception Module** that translates raw sensor data into symbols or observations the planning engine can use.

- **LLM for Task and Motion Planning:** Cutting-edge research is exploring letting large language models handle planning for robotics tasks. In some experimental frameworks, an LLM generates not only high-level plans but also low-level motion commands directly (this is referred to as LLM+Affordance in recent literature) [19]. For example, an LLM-based system might output a sequence of robot joint movements after reasoning in natural language [19]. However, in practice, it's often safer to keep the LLM at the strategic level and use dedicated motion planners for precise control. PulZ can implement a hybrid: the AI agent decides *what* the robot should do and *in what order* (e.g. "pick up crate A from location X, then move it to Y"), but a lower-level controller (or a script) executes the *how* (path planning, inverse kinematics) for each step. The agent would then

monitor execution via sensor feedback and, if an unexpected situation arises, re-plan or prompt for human intervention. This aligns with an **indirect control approach** for safety [20] – the LLM agent stays the brains of the operation, while proven robotics software handles real-time control loops.

- **Simulation and Digital Twins:** To further ensure safety before the AI acts in the real world, you could incorporate simulation testing. As of 2025, many robotics deployments use digital twin environments or physics simulators. PulZ could integrate with a simulator (e.g. a Gazebo or Webots environment for the warehouse scenario) as a tool. The agent can run a "dry-run" of the plan in sim and analyze the outcome, which adds an extra validation layer before physical execution. This kind of simulated preview would help the system **learn from machine input** in a safe setting – e.g. if the simulation shows boxes toppling, the agent can adjust the plan or request a different tool.

- **Real-time Adaptation:** Unlike purely digital tasks where API calls either succeed or fail quickly, physical tasks are continuous and time-extended. PulZ will need a mechanism for **real-time monitoring and dynamic adjustment** during a task. Your architecture already has an Execution Monitor concept [21] [22]; extending that for robotics might involve tighter feedback loops. For instance, the agent could break a long physical task into micro-steps and after each movement, evaluate sensor data (within the `Execution Monitor`) to decide if it should proceed, modify the plan, or abort. By 2026 there's emphasis on such feedback-based planning in robotics AI – essentially combining deliberative planning with reactive control. PulZ is conceptually well-suited for this: the agent can be in a loop of *plan* → *act* → *sense* → *re-plan*, which is exactly what "learning from machine input" implies. This also ties into cross-session learning: the system should store what it learned (e.g. "box type A is heavier than expected, use two arms or reduce speed next time") in its memory for future runs.

In summary, **to not only cover business automations but also "beyond" (physical world)**, PulZ should embrace multimodal and robotics integration. This means adding components for **perception**, defining hardware interfaces as tools, and incorporating robust real-time feedback control. The good news is your base architecture (modular tools, planning with validations, etc.) is adaptable to this – it just needs plugins for vision and robotics. By doing this, PulZ would join the ranks of cutting-edge systems that bridge LLM reasoning with embodied action, a frontier of AI in 2025-2026.

📈 Developments in Autonomous Agent Systems (2024–2026)

The AI landscape is rapidly evolving, and it's wise to **update PulZ with the latest best practices and innovations**. Here are relevant high-end developments up to 2026 that you might consider:

- **Rise of Agent Orchestration Frameworks:** Over the past two years, a number of frameworks emerged to help build autonomous agents. For example, **SuperAGI** is an open-source platform specifically for orchestrating multi-agent workflows [23]. It allows chaining agents and tools, and even provides a marketplace of ready-made skills [24]. Similarly, enterprise offerings like **IBM watsonx Orchestrate** and **UiPath's AI Orchestrator** combine LLM reasoning with business process automation [25] [26]. The **good news**: the core ideas in these platforms (integrations, tool libraries, etc.) are already present in PulZ's design. PulZ's differentiator could be its unified approach (where others might require more manual configuration) and its strong safety/learning focus. Nonetheless, staying aware of these will help you **position PulZ**. It might make sense to ensure PulZ can integrate

with or import tools from such ecosystems (e.g., using existing tool definitions from SuperAGI's marketplace) to leverage what's already out there.

- **Multi-Agent Collaboration:** Some cutting-edge systems deploy multiple specialized agents that communicate with each other to solve complex tasks (for instance, an "Engineer Agent" writing code and a "Evaluator Agent" checking it). As of 2025, research prototypes like AutoGPT with multiple agents or OpenAI's experiments with agent dialogues gained attention. PulZ's architecture currently centers on one intelligent agent orchestrating everything, which is a simpler and often more efficient approach (less overhead in coordination). However, you could incorporate **multi-agent capabilities** in the future: e.g., the system could spin up a second planning agent to double-check plans (for added safety), or spawn domain-specific agents (a vision-specialist agent, a financial-calculation agent, etc.) and then merge their results. If this path is of interest, ensure the communication infrastructure is flexible – e.g., a messaging or memory-passing interface for agents. The **Shakudo AgentFlow** platform is an example that wraps multiple agents in a coordinated workflow graph [27]. While multi-agent setups are not always needed, being able to do "a series of systems in sync" could extend PulZ for highly complex problems or enterprise-scale deployments.

- **Improved Memory and Knowledge Integration:** By 2026, we've seen better techniques for an agent to possess long-term knowledge. Vector databases (like the Qdrant integration you planned [28]) remain popular for semantic memory. In addition, **knowledge graphs** are making a comeback for storing structured facts the agent learns, and new retrieval techniques allow an agent to dynamically query databases or the web in real-time. PulZ might benefit from a **hybrid memory**: use vectors for unstructured context and a relational or graph store for key facts/outcomes. The system could then answer or plan with a combination of "remembered" lessons and fresh info from live queries. Ensuring PulZ can plug into enterprise knowledge bases or IoT data streams (for physical deployments) will make it more powerful. Fortunately, your design's modular memory and tool layers will support this – it's more about configuring additional connectors (e.g., a connector to an internal database, or to a real-time data feed).

- **Advances in Tool Use and API Integration:** A notable trend is standardizing how AI agents discover and use tools. OpenAI's **function calling** feature (introduced mid-2023) became a de-facto method for tools: developers define functions with schemas that the LLM can call. Your plan for "self-documenting tools" is very much in line with this movement [17]. By 2025, we also have more **open standards** like PAPILLON (hypothetical example for tool description) or simply widespread use of OpenAPI/Swagger specs to let agents autonomously understand APIs. PulZ should continue to embrace **open integration standards** – e.g., it could read an OpenAPI spec file of a new service to auto-generate a tool interface. This avoids getting locked into proprietary formats and makes it easy to add/swap tools. *(Indeed, one of the key factors businesses look for in orchestration platforms is modularity and extensibility – the ability to add or swap models and tools easily [29]. PulZ's design as a flexible orchestrator fits this requirement perfectly.)*

- **State-of-the-Art Models and Reasoning**: Since your system is model-agnostic, you'll want to slot in the best models available. As of 2026, the landscape likely includes **GPT-4/5**, **Anthropic Claude 2 or beyond**, **Google Gemini (multi-modal)**, and strong open-source models (Llama 3/4, etc.). Keep an eye on the latest LLMs for upgrades. The instruction pack already suggests using a cost-effective model for most work and a more powerful one for tricky cases [30] [31] – that strategy is even more relevant now with new models emerging. For example, you might use a local Llama-3 70B for

inexpensive reasoning and only call an external GPT-5 for very complex or crucial steps. Also, consider specialized models: by now there are likely better code generation models, better vision models, etc. The **PulZ system should allow plugging these in as needed**, which brings us to modularity:

Modular Model Integration and Expandability

Your point about swapping models and using paid APIs once the system is running is crucial. PulZ should be designed as a **model-agnostic orchestrator**:

- **Abstraction Layer for Models:** Implement a clear interface for language models (and other AI models) so that the underlying provider can be changed with minimal effort. For instance, a `LLMService` class could have implementations for OpenAI API, Anthropic API, local HuggingFace model, etc. At runtime or via config, one can select which to use. This way, if a new high-end model appears, you can plug it in by writing a new connector rather than redesigning the system. This was already hinted in your docs by listing multiple LLM options (DeepSeek, Claude, Llama) [32]. Make sure this is a formal part of the architecture.

- **Tool and Skill Plugins:** Similarly, treat tool integrations as plugins. The system can have a registry of tool definitions (some might be local scripts, others hit external APIs or hardware). Users of PulZ (like different departments in a company, or different deployment contexts) should be able to **enable or disable sets of tools/peripherals** easily. For example, at the 3D printing company, enable CAD design tools and printer controls; for an office workflow, those are off but CRM and email tools are on. A well-defined JSON/YAML format for tool specs (as you have) will allow this kind of customization simply by adding files or entries, without changing core code.

- **Model Upgrades and Ensemble Approaches:** By 2025, a lot of orchestration systems started using **multiple models in tandem** – not just one "brain". For instance, an agent might use a large general model for reasoning but a smaller code-focused model for writing code, or a math-focused model for calculations. PulZ can incorporate this idea: route a task to a different model if appropriate. Your strategy of using a cheap model 90% of the time and a expensive one selectively is a form of this [33]. Expand on it by including **specialist models**. E.g., for image analysis steps, use a vision model; for speech, use an audio model. The orchestrator (the PulZ brain) can decide or be configured which model to use per tool or per step. This keeps costs down and performance up.

- **Avoiding Vendor Lock-In:** As noted, one priority is not getting stuck with one AI vendor. The **best practice** here is to support open standards and self-hosting. For instance, using the **ONNX** format for models (if you ever export models or use runtimes) ensures interoperability [29]. Also, by having local model support (which you do plan via Ollama + Llama) [34], you maintain leverage when negotiating with API providers. In 2025, many companies valued this flexibility; it's wise for PulZ to continue with that philosophy.

Overall, PulZ's architecture should be **modular at every level** – models, tools, memory stores, UI, etc. This was in your design ethos and it remains vital. It will allow the system to evolve with the AI landscape. A practical suggestion is to clearly organize the repository or code into modules (for example: `core/` for core planning logic, `models/` for model adapters, `tools/` for tool definitions, `memory/` for memory

backends, etc.), so that each part can be extended or replaced independently. (You mentioned “no folders” for now, but conceptually keep this modular structure in mind as code grows.)

Project Initialization & Next Steps in the Repo

Now that you have access to the GitHub repo, here’s how you might **kick off the project** (in terms of laying out key components, without diving into full code yet):

1. **Documentation & Planning:** Start by populating a **README** or an **Overview.md** in the repo that summarizes the project vision, scope, and architecture. You can adapt content from your “PulZ Project Overview” and the points above into this document. This sets the stage for collaborators and makes it clear what the project is about. Include a high-level diagram of the architecture if possible (even an ASCII or simple block diagram for now) – illustrating the Planning Layer, Memory, Tool interfaces, and Safety components working together. Given the richness of your PDFs, consider adding a docs/ folder later (but as you said, perhaps hold off on complex structure until initial commit is done).

2. **Core Orchestration Module:** Set up the basic scaffolding for the core system – for example, if you’re using Python, create a package like `pulz_core` with placeholder classes: `GoalParser`, `StrategyGenerator`, `Executor`, `Validator`, etc. If Node/TypeScript (since you mentioned Supabase Edge Functions, TS might be in play), you could start a Next.js or Node project and plan out similar classes or modules. The idea is to lay down the **skeleton** of the system: it doesn’t function fully yet, but we know the main pieces. This also helps identify any immediate integration needs (e.g., a config file for API keys, or a connection to Supabase).

3. **Tool/Peripheral Schema Definition:** Add a template (perhaps a JSON schema or TypeScript interface) for how tools will be defined. You might create a `tools/` directory (when ready) or simply a couple of example tool definition files (e.g., `web_search.tool.json`, `robot_arm.tool.json` as dummy examples). These will document how the system expects to describe actions, inputs, outputs of tools. Down the line, the agent will parse these to use the tools. Having a few examples now (even if not implemented) will be useful for testing the parsing logic and demonstrating extensibility.

4. **Memory and Database Setup:** Since you intend to use Supabase (Postgres) for persistent storage, you can start writing the SQL schema (as you partly did in the design prompt). Consider creating a `schema.sql` or using Supabase migration files. Define tables for Executions, Steps, Tools, etc. For vector memory, integrate Qdrant or another vector DB (you might delay this until you have something to store). At least, ensure your code scaffolding has hooks for memory: e.g., a class `VectorMemory` with methods like `store_embedding(text)` and `query(similarity)`. Initially, these can be stubs or even use an in-memory list, but structuring it now helps later when you plug in the real DB.

5. **Frontend/UI Stub:** If this is full-stack, you might also scaffold the UI (especially since you have a strong vision for it: chat interface, progress bar, etc.). Setting up a basic Next.js project with Shadcn/UI, and a simple page that has a textarea and a “Execute” button, would be a nice quick start. It won’t *do* much until the backend is wired, but you can commit the boilerplate. This signals the direction

and lets others (like your collaborator or mentor) easily run a minimal interface. You can even hard-code a fake "Hello, world" response just to show the plumbings. Again, keep UI in a separate module (or at least clearly separated from core logic) so the core can potentially run headless as well (for API use, etc.).

6. **Initial Testing Approach:** As you lay out code, set up a testing framework (even simple unit tests). For example, write a test for the GoalParser (feeding it a sample goal and seeing if it produces a structured objective). This will enforce that each piece is working as intended. Given the complexity, having a habit of writing tests will help down the road. You don't need many now – just enough to validate the skeleton (like, does the orchestrator class call the parser and then the strategist in sequence).

By starting with the above, you effectively create the **backbone** of the project in the repository. No deep folders are needed immediately – perhaps just a handful: one for core logic, one for any serverless functions (if using Supabase edge functions, those might live in a `functions/` directory), and one for the web app if applicable. You can keep it flat to begin with and refactor into subfolders as it grows.

Also consider adding a license file at this stage (discussed next) and a contributing guide if you plan to open it to the community. Early clarity will attract interest, especially since you might build a community around it.

Licensing Considerations

Choosing the right open-source license is important, especially if you plan to build a community and possibly commercialize parts of the system (e.g. hosting service). The **good news** is that your open-source strategy (open core, paid hosting) is common, and the licenses typically used are **permissive ones**. The two most popular in 2025 have been **MIT and Apache 2.0** [35] :

- **MIT License:** A very permissive, simple license. It basically says anyone can use, modify, distribute your code, with no warranty, as long as they include the license notice. MIT is extremely popular for startups and open-source projects due to its simplicity and few restrictions – in fact, it was the #1 most used license in 2025 [36] . If you want maximal adoption and minimal fuss, MIT is a great choice. It allows commercial use (so you or others can build on it freely). One thing to note: MIT **does not include an explicit patent grant**. So if you have any patented techniques in PulZ, MIT doesn't automatically give users rights to those – which can be either a pro or con depending on your strategy (it means you retain the ability to enforce patents separately).

- **Apache License 2.0:** Apache 2.0 is also permissive like MIT, but it's a bit longer because it includes extra protections. Notably, Apache 2.0 **explicitly grants patent rights** to users of the code [37] . This is reassuring for users and enterprises because they don't have to fear a patent lawsuit from contributors. Apache also has requirements like stating changes and not using trademarks, but generally it's quite friendly. It was the #2 most popular license by 2025 [38] , and is favored in enterprise and foundation contexts (many big projects use Apache 2.0 for the legal clarity). If you anticipate external contributors or wide enterprise adoption, Apache 2.0 is a solid choice – it "protects developers and users alike" by covering patent use [37] .

- **Other licenses:** You likely *don't* want a copyleft (GPL) license here, because that would force anyone who uses your code to open-source their whole project, which could limit adoption and conflict with your SaaS model. Your documents even suggest the core being MIT [39], which implies a permissive approach. So it's really MIT vs Apache in spirit. Both let people use PulZ freely; Apache just adds the patent language and some extra legal boilerplate. There's also BSD, but that's similar to MIT (and less common these days compared to MIT/Apache). Given the trend, sticking with one of the top two makes sense.

Recommendation: If you're okay with the slight complexity, **Apache 2.0 might be the best fit** for PulZ. It encourages adoption but also signals to companies that they're safe from patent claims (important if you want, say, an investor or partners to feel comfortable using the code). Apache is often chosen for frameworks that could become industry standards, which PulZ aspires to be. On the other hand, if you want ultimate simplicity and to retain more control over patents, **MIT** is perfectly fine and very well-understood. Many startups choose MIT to maximize usage and because they plan to offer proprietary services on top (which is aligned with your open-core idea). In fact, your own strategy note explicitly listed the core as MIT-licensed [39], which is a strong vote for MIT.

So, you **can't go wrong with either MIT or Apache 2.0** – they are by far the dominant choices in modern open source (together they account for the majority of OSS licenses in use) [35] [38]. If pressed: - Choose **MIT** if you value simplicity and might want the option to enforce patents or proprietary add-ons separately. - Choose **Apache 2.0** if you want to foster a contributor community with maximum legal clarity and don't mind granting patent usage to all users up front.

In any case, include a `LICENSE` file in the repo root with the full text of the chosen license. Also put a short blurb in the README ("Licensed under the MIT License" or "Apache License 2.0") so it's obvious.

Conclusion

In summary, your instruction pack is comprehensive and **already aligns with state-of-the-art concepts** for AI orchestration. The additions we discussed – integrating multi-modal/robotic capabilities, staying abreast of new agent frameworks, ensuring modular upgradability, and structuring the project for growth – will position **PulZ** at the cutting edge in 2026.

Your vision of a series of systems working in sync, learning from each other and from the world (digital or physical), is very much in line with where AI as a whole is headed. By proceeding with careful design and incorporating these latest developments, PulZ could truly become a powerful generalist AI orchestrator.

Moving forward, focus on incremental build-out: get a prototype running with core features, then iterate by adding the fancy bells and whistles (advanced models, robot integration, etc.) one by one. Given the solid groundwork in the instruction pack and the additional research here, you have a clear roadmap.

Good luck with pushing this project to the repo and beyond – it's an exciting endeavor, and with the right execution, PulZ could be one of the high-end AI systems that defines this era!

Sources:

- PulZ System Architecture and Design Documents [2] [9] [4] (provided instruction pack)
- Zeng et al., *"A Survey on LLMs in Robotics"* – discussing integration of language models with robotic planning and control [40] [41]
- Domo Research, *"10 Best AI Orchestration Platforms in 2025"* – industry trends on orchestration tools and the importance of integration & extensibility [42] [23]
- Open Source Initiative & Linuxiac – statistics on open-source license popularity in 2025 (MIT & Apache 2.0 leading) [35] [38]
- Mend.io Blog – explanation of Apache 2.0 license's patent clause and its enterprise popularity [37]

---

[1] [4] [12] [13] [28] [30] [31] [32] [33] [34] [39] Self-Orchestrating AI Agent System - Research & Patent Documentation.PDF
file://file_000000002ee871f7947b04d702f595d9

[2] [3] [5] [6] [7] [8] [9] [10] [11] [14] [15] [16] [17] [21] [22] Next-Gen AI Agent System Architecture.PDF
file://file_00000000bfec71f5942fb69fc6a63924

[18] [40] PaLM-E: An Embodied Multimodal Language Model | Request PDF
https://www.researchgate.net/publication/369035918_PaLM-E_An_Embodied_Multimodal_Language_Model

[19] LLM+A: Grounding Large Language Models in Physical World with...
https://openreview.net/forum?id=cbVnJa4l2o

[20] [41] A Survey on Integration of Large Language Models with Intelligent Robots
https://arxiv.org/html/2404.09228v5

[23] [24] [25] [26] [29] [42] 10 Best AI Orchestration Platforms in 2025: Features, Benefits & Use Cases
https://www.domo.com/learn/article/best-ai-orchestration-platforms

[27] Top 9 AI Agent Frameworks as of December 2025 | Shakudo
https://www.shakudo.io/blog/top-9-ai-agent-frameworks

[35] [36] [38] MIT and Apache 2.0 Lead Open Source Licensing in 2025
https://linuxiac.com/mit-and-apache-2-0-lead-open-source-licensing-in-2025/

[37] Top Open Source Licenses Explained
https://www.mend.io/blog/top-open-source-licenses-explained/