ChatGPT

Interconnected Exosystème – Complete Downloadable Bundle

The **Interconnected Exosystème** bundle provides a comprehensive package of the project's code, documentation, configuration, and visual simulation. It is organized with a minimalist structure, reflecting the project's ideology of blending art, science, and technology in a fractal, self-similar network. Below we outline each element of the bundle – including the directory layout, README, whitepaper, interactive simulation script, config files, and licensing – all presented in a clear, technical yet symbolically-informed style.

Project Ideology and Overview

The project is founded on a cross-disciplinary **ideology** that unites low-level technical rigor with high-level creative expression [1]. It treats the entire system as a living **"digital ecosystem"** or *"cyber tribe,"* where each part (code module, domain name, agent) is distinct yet interconnected [2]. In practical terms, this means the software, hardware, and conceptual layers are intertwined in a hyperbolic, fractal-like structure – networks within networks – aiming to reduce entropy through emergent order. The project's documentation describes an *"open playground"* that *"shares tools to bridge creative coding, science and music"* while pursuing experiments that are *"simultaneously aesthetic and scientific"* [2]. In other words, it is as much an art project as an engineering endeavor, where **code becomes poetry and domains become domains of knowledge** [3].

Symbolic archetypes play a key role in this ecosystem. Major components are personified by codenames like **Kobalt**, **Phoebii**, **Uniphi**, etc., each representing an aspect of the system's character. For example, *Kobalt* appears as an author of generative algorithms and is regarded as a foundational pillar – the structured, knowledge-bearing "engine" of the system [4]. *Phoebii*, by contrast, alludes to a bright, artistic persona, embodying the imaginative or visual facet of the project [5]. These archetypes bind symbolic meaning to real system functions [6], ensuring that every module is not just a piece of code but also part of a narrative **(abstract metaphysics meets group theory)**. This design philosophy yields a **modular mesh** of agents and services that operate like an orchestra – distinct instruments following a common score [3].

In summary, the project's vision is an **"Afrhônet Uniphil Interconnected Exosystème,"** a holistic network of networks. It blends concepts from thermodynamics (entropy and self-organization), information theory (data networks, DNS namespaces), and philosophy (symbolic meaning) into one continuous feedback system [7]. This ethos is carried through every element of the bundle, from the documentation tone to the engineering of the simulation.

Bundle Contents and Structure

The downloadable bundle is packaged as a ZIP file containing all the necessary components. The directory structure is kept minimal and logical for clarity:

- `README.md` – A concise introduction to the project, usage instructions, and the philosophical context.

- `docs/` – Documentation including the project **whitepaper**, technical papers, and concept write-ups (e.g. `whitepaper.md`, architecture overviews).
- `src/` **or** `app/` – Source code for the interactive simulation and any supporting libraries:
- *Visualization Script* (e.g. `exosystem_3d.html` or `index.html`) – The main HTML/JS file implementing the 3D force-directed graph world (details in next section).
- *Shader Files* (if any, e.g. `shaders.glsl` or inline in the HTML) – Custom WebGL shaders for rendering blobby spheres and effects.
- *Data/Config Files* – JSON or JavaScript files defining network parameters, initial graph data, and system settings (see **Configuration** below).
- `config/` – Configuration for deployment and environment:
- For example, network outpost settings (mirroring what was in the "umowt" config – FTP hosts, domains, etc.) ensuring the simulation can connect or represent real endpoints [8].
- Simulation tuning parameters (could also be in a JSON as noted above).
- `assets/` – (If applicable) Any static assets like images, fonts, or audio used by the UI.
- `.kobalt/` – (If present) Special project files or templates (e.g. prompts or generative script templates).
- `LICENSE` – License documents for the project (see **Licensing** section).
- `README-*.md` – Additional READMEs or guides (for sub-modules, if needed – e.g. a frontend or backend guide if those are included).

This structure keeps content organized by function: documentation vs. source vs. config. The **README** at the root provides an entry point for anyone downloading the bundle, while deeper information is in the docs. Next, we delve into the key elements:

README: Minimalist Project Guide

The `README.md` is intentionally **succinct and clear**, following a minimalism ethos. It outlines:

- **Project Purpose & Philosophy:** A brief statement of what the project is (e.g. *"An open-source creative tech ecosystem blending generative code, network simulation, and interactive art"*), and a note on the core philosophy (bridging creative coding and science as discussed above). This might include a one-line motto or the © 2025 Kobalt credit line (as seen in the PWA footer) to reinforce the creative persona [9].
- **Contents:** A list of bundle components (similar to the structure above) so users know where to find things.
- **Quickstart Usage:** Simple instructions to run the visualization (e.g. "open `exosystem_3d.html` in a WebGL-capable browser" or run a local server if needed), and how to access the UI controls.
- **Project Status:** Notes on the current stage (sandbox/experimental), and pointers to further documentation (like *"See the whitepaper in docs/ for a deep dive"*).
- **Credits:** Acknowledgments of the contributors or the archetypal agents (for fun, it might list *Kobalt, Phoebii,* etc. as contributors along with real names).
- **Footer:** The tagline (e.g. *"© 2025 Kobalt – Exploration des écosystèmes créatifs"*) reaffirming the creative exploration mission [10].

The tone of the README remains **technical but inspirational**, encouraging exploration. It avoids clutter – using bullet points and short paragraphs – to embody the bright, concise style of the project.

Whitepaper and Documentation

In the `docs/` folder, the **whitepaper** (e.g. `whitepaper.md` or PDF) provides an exhaustive yet cohesively written description of the project's theoretical framework and architecture. It likely includes:

- **Abstract:** High-level summary of the project's goals (blending art and science, multi-layer network concept, etc.).
- **Background & Motivation:** Discussion of the inspirations – from Mohist philosophy to ecosystem science – that led to the project's creation [11] [12]. It explains the need for an open playground for creative tech, citing how domain names, servers, and agents form a *"semantic network"* bridging human ideas to machine representations [13].
- **Architecture:** Description of the system's layers and agents. The whitepaper outlines how low-level components (transistors, code) build up to high-level behaviors (AI, creative outputs), mapping each archetype to a layer. For instance, Kobalt as core logic, Phoebii as creative module, Ūmöwt as deployment engine [4] [5]. It might illustrate how a domain name (like `kobalt.io` or others) corresponds to modules, reinforcing intellectual property clarity around those names [14] [15].
- **Network Simulation Model:** Mathematical and conceptual explanation of the force-directed meta-graph. This section ties directly into the code: it may describe nodes, paths, and networks and reference **group theory** analogies (each network of nodes could be considered a "group" with symmetric interactions). It might even include figures or formulas for the force calculations (repulsion, attraction, alignment forces, etc.).
- **Results & Screenshots:** If available, images of the visualization (e.g. 2D prototype shots from `exotil.html`) showing clustered blobby networks. These illustrate how the system *"translates human ideas (art, philosophy, science) into machine-interpretable forms and back into human experiences (visual art, interactive media)"* [16] [17].
- **Conclusion & Outlook:** Reflection on how the project demonstrates a *"holistic approach to blending mediums and fields"*, with code, data, and visuals interlinked [18] [19]. It likely reiterates the vision of *"a 21st-century art-science collaboration"* and suggests future improvements (e.g. scaling up the simulation, integrating live data, etc.).

The writing style of the whitepaper is more narrative and philosophical than a typical engineering doc, aligning with the project's unique ethos. It still provides practical technical content (so engineers can follow the implementation), but it does so in an engaging, metaphor-rich language. For example, it might describe the system as *"a grand interdisciplinary symphony"* where each component *"plays a role much like instruments in an orchestra"* [3] – poetic yet precise.

3D Network Visualization Script (Force-Directed Graph)

One of the highlights of the bundle is the **interactive 3D network visualization**, implemented in a single-page web application (HTML/JavaScript). This script is the evolution of earlier 2D experiments (`exosphere_2.html`, `exotil.html` prototypes) into a GPU-accelerated 3D world. It creates a **force-directed graph** where nodes form blobby spheres, connected by path links into clusters, and multiple clusters interact in a meta-network.

Technology stack: The visualization uses **vanilla JavaScript** alongside the WebGL API (no heavy frameworks, to stay minimal). It harnesses GPU shaders for rendering efficiency. The choice of vanilla JS ensures low-level control over performance and visuals, while WebGL provides the raw power to handle potentially hundreds or thousands of particles in real-time. According to the architecture notes, the dashboard technology includes D3.js and WebGL [20]; however, here the approach is to use custom JS for physics and WebGL for rendering, keeping external dependencies minimal.

Data & Initialization: The graph's initial state can be defined in a JSON configuration (either embedded or in a separate file). For example, a simplified `network.json` might look like:

```
{
  "networks": [
    {
      "id": 0,
      "paths": [
        {
          "id": 0,
          "nodes": [
            { "id": "0-0", "position": [0, 0, 0], "radius": 1.5 },
            { "id": "0-1", "position": [1.2, 0.5, -0.3], "radius": 1.5 },
            { "id": "0-2", "position": [-1.0, -0.8, 0.2], "radius": 1.5 }
            /* ...other nodes forming a closed path... */
          ]
        }
        /* ...other paths in network 0... */
      ]
    }
    /* ...other networks... */
  ],
  "parameters": {
    "nodeCount": 100,
    "pathCount": 10,
    "repulsionForce": 0.5,
    "attractionForce": 0.2,
    "timeStep": 0.016
  }
}
```

This illustrates the concept: each **network** contains multiple **paths**, each path contains several **nodes** with 3D positions. The `parameters` section defines global physics constants and initial counts (if the script generates additional nodes procedurally). The actual bundle's default data might be more complex (and likely generated procedurally rather than hand-coded positions), but providing it in JSON form makes it easy to tweak and also aligns with the project's theme of representing complex systems in data structures [21].

Physics simulation: Once initialized, the script runs a **force-directed simulation** on each frame: - Nodes have properties like position, velocity, acceleration, mass, etc. - **Forces:** The system applies several forces each tick: - **Repulsion:** Nodes push away from each other (or paths push other paths) to maintain spacing (avoiding overlap). For example, each node pair could exert a Coulomb-like repulsive force that falls off with distance squared. - **Attraction:** Connected nodes (neighbors along a path) and neighboring paths have spring-like attractive forces to hold clusters together. Path loops might be kept in shape by linking each node to its next neighbor and perhaps the first/last forming a closed loop. - **Alignment/Cohesion:** There may be forces to keep paths aligned when they belong to the same network cluster – akin to flocking behavior (steering towards the cluster's center or matching velocity). - **Global Centering:** All networks are gently pulled toward the world origin (0,0,0) to keep the entire graph in view and encourage clustering (lessening entropy by avoiding runaway dispersion). In code,

this is done by finding each network's center of mass and applying a vector toward the origin [22]. - Optionally, **damping** or drag is applied via velocity adjustments to stabilize the movement over time.

The earlier 2D code (`exotil.html`) implemented many of these concepts. For instance, it computed each network's center of mass and diameter, and applied attraction of networks to the canvas center [22]. It also allowed **networks to exchange paths** – a path (node loop) could hop from one network cluster to another, altering the cluster structure dynamically [23]. This novel feature creates a meta-graph behavior: clusters merging or splitting over time, simulating an ever-evolving fractal network of networks.

In the 3D version, these dynamics persist but in three dimensions. The code uses basic vector algebra (extended to 3D) for forces. Linear algebra utilities (Vector3 class with add, sub, normalize, etc.) are likely defined to simplify calculations.

Rendering: Using WebGL, each node is rendered as a **sphere** (or point sprite) and each connection as a tube or line: - To achieve the **"blobby"** effect where connected nodes blend into each other, a common technique is to use metaballs or smooth particle rendering. For example, in the fragment shader, each node can be treated as an influence field (an isosurface of a scalar field). When two nodes are close, their fields merge to render as one continuous blob. This can be done by summing contributions of nearby nodes and drawing an iso-contour at a threshold. A simpler approach is to use **additive blending**: draw semi-transparent spheres so that overlapping areas appear brighter/denser, visually suggesting a connection. - Each path could be given a distinct color (perhaps assigned per network or per path), helping to distinguish clusters. In the previous 2D code, there were references to `paths[i].color` in drawing loops [24], implying each path had its own color. - The background is likely dark or neutral to highlight the colorful node clusters (common in such visualizations).

The **vertex shader** would position each node sphere in 3D (using the node's position array), and a **fragment shader** could handle lighting or the metaball blending if implemented. Given minimalism, lighting might be very basic (even just a simple ambient + directional light for shading the spheres).

Camera & Interaction: The world can be navigated in 3D – typically the user can rotate, pan, and zoom the view (via mouse or touch input). A simple trackball or orbital camera control can be implemented in vanilla JS to allow exploration of the graph from different angles.

Real-time updates: As the physics runs, the visualization updates at, say, 60 FPS. The nodes move according to forces, and the user sees clusters form, merge, or drift. The simulation effectively demonstrates a **fractal-like lessening of entropy**: random initial positions settle into more organized clusters due to the attractive forces and exchanges. Over time, one might see a higher-level structure emerge (for example, multiple clusters orbiting each other or forming a lattice).

Interactive UI Panel and Controls

To give users insight and control over the simulation, the bundle includes an **interactive UI panel**. This is typically a collapsible sidebar or overlay with various controls and real-time data displays. In line with the project's **transparency and user empowerment**, all significant parameters are exposed for tweaking.

Key features of the UI panel:

- **Simulation Parameters:** Sliders or input fields for all force constants (repulsion strength, attraction strength, etc.), node count, path count, cluster count, gravity centering force, time step, and any other tunable values. This allows live tuning; for example, increasing repulsion might cause clusters to disperse more, while increasing attraction might tighten them.
- **Play/Pause & Step:** Buttons to pause the simulation or step frame-by-frame, useful for examining a static state or debugging the dynamics.
- **Add/Remove Elements:** Controls to add a new network cluster or remove one on the fly. This leverages functions like `generateNetwork()` or `removeNetwork()` from the code (which existed in 2D form) to dynamically grow or shrink the meta-structure.
- **Statistics:** Real-time readouts such as current number of nodes, paths, networks; perhaps average cluster size; maybe an entropy metric (since a *ValidationAgent* in the backend monitors entropy [25], the frontend might similarly display a measure of disorder vs order).
- **Visual Toggles:** Options to toggle rendering modes – e.g. switch between blobby sphere view and a simple wireframe graph, or enable labels on nodes, etc. One could also toggle layers of detail (see **Multi-layer view** below).

The panel is likely implemented in plain HTML/CSS with some lightweight JS (or possibly using a tiny library like dat.GUI for convenience). It starts hidden or semi-transparent and can be expanded by the user when needed, so as not to clutter the screen.

Dynamic Inline Editing: A standout feature is the ability to click on any network element in the visualization and get an **inline detail editor**: - If a **node** is clicked: the panel could show that node's properties (ID, current coordinates, velocity vector, maybe what real-world entity it represents if any). The user might be able to adjust certain properties (e.g. mark it as fixed in space, or assign it to a different cluster). - If a **path (edge group)** is clicked: the panel shows that path's info, like which nodes belong to it, its length or circumference, maybe the color. The user could, say, change the path's color or remove a specific node. - If a **network/cluster** is clicked (perhaps by clicking near the center of a cluster or selecting a cluster from a list): the panel enumerates all paths in that network, allows renaming the cluster or merging it with another.

This inline editing essentially treats the simulation as an open-ended sandbox, where one can drill down from the highest level (*cluster of clusters*) to the lowest (*individual node*). The UI might use a tree view or breadcrumb to indicate the nesting: e.g. **Network 2 > Path 5 > Node 5-3** to show context of a selected node.

All changes made through the panel are applied live to the running simulation. For instance, editing a node's position will immediately move it in the WebGL scene (and the physics engine might then respond with forces adjusting around the new position).

Multi-Layer Semantic View: In line with the project's concept of encapsulated layers (electron → transistor → logic gate → neural network), the UI could offer **layer filters**. This means the graph can be interpreted at different abstraction levels: - **Physical layer:** Nodes represent fundamental units (electrons or hardware pins), paths represent physical connections (wires, circuits), networks might represent hardware components. This layer would have a very large number of nodes if fully realized, so it might be more conceptual unless integrated with actual hardware data. - **Logical layer:** Nodes represent higher-level components like processing units or routers, paths are communication links, networks are subnets or modules. This could correspond to a network topology view. - **AI/Neural layer:** Nodes represent neurons or perceptrons, paths represent neural connections, networks are entire neural nets or modules in a deep learning system. Indeed, the **flexible perceptron** model (referenced

as `flexibleperceptiron.html`) suggests the system can simulate neural network behavior. Potentially, one cluster of nodes could be toggled to behave as a self-improving perceptron group. - **Creative layer:** At the highest level, each network might represent an "agent" or persona (Kobalt, Phoebii, etc.), and links between them show collaborations or data flows. This is a very abstract view, but it closes the loop by tying the tech back to the personas.

The UI might allow switching the **labeling** of nodes/edges according to these layers. For example, a node could have multiple identities (one for each layer). Clicking a node might reveal: *"Node ID 42 – as electron X in transistor Y; as neuron #5 in layer 2; as agent Kobalt's sub-node,"* etc., demonstrating nested identities. This fulfills the idea of **"showing every encapsulated, nested element from an electron through transistor to complex routing logic to deep learning neural networks built by perceptron clusters."** In practice, not every layer may be fully simulated (the current sandbox might focus on the network/neural level), but the architecture is **flexible** enough to extend or reinterpret the model at different scales.

Overall, the interactive UI ensures the system is not a black box – the user can **observe, manipulate, and validate** every part of the network. This reflects the project's ethos of openness and exploration, giving a mindful awareness of how small changes ripple through the system (truly *"mindful mind"* design).

Configuration and Flexibility

The bundle's configuration files enable customization without altering code. Key configuration aspects include:

- **Simulation Config (JSON):** As mentioned, parameters for the physics and initial topology can be tweaked via a JSON or JS config. Users can edit this file before running the simulation to change the number of nodes or the strength of forces globally. This is useful for running the system in different modes (e.g., a dense graph vs. a sparse one) or for loading a specific saved network state.
- **Deployment Config:** The project likely has config related to how it deploys or mirrors content across the internet (since the codebase integrates a *"self-sovereign namespace"* logic [26]). For example, there may be a config file enumerating domain names (like `kobalth2.free.fr`, `phepsilon.free.fr` as mentioned in an internal config [27]) and credentials for FTP or IPFS deployment. These would reside in `config/` and guide the **backend or devops** side (e.g., how updates to the simulation or content are pushed to various servers).
- **Agent Settings:** If the simulation ties into actual running agents (backend processes named Uniphi, SyncAgent, etc.), there could be a config file specifying their endpoints, ports or toggles to simulate their presence. However, given this is a sandbox bundle, those might be stubbed out or documented rather than active.
- **Licenses & Modular Structure:** In a sense, the licenses chosen (next section) are also part of the configuration of the project's openness. There might be a `LICENSES` folder if multiple licenses apply, or a section in the README explaining how different parts are licensed.

The configuration design follows **minimalism and clarity** – using standard formats (JSON, .env files, Markdown) that are easy to read and edit. This way, technically adept users can fine-tune the exosystem, and non-technical collaborators (artists, theorists) can also adjust high-level settings or content (like domain names or titles) without digging into code.

Licensing Strategy

Reflecting the project's open and interdisciplinary ideology, the bundle explores a **modular licensing** approach. The goal is to balance freedom and collaboration with clarity and legal assurance [28]. Here are the key components of the licensing strategy:

- **Source Code License:** The core simulation code and associated scripts are released under an **open-source license** to encourage use and contributions. A strong choice is the **MIT License**, which is permissive and widely understood – fitting the idea of sharing tools openly. MIT allows anyone to reuse the code in creative coding communities with minimal friction. As an alternative, if the project maintainers want to ensure improvements remain open, they might opt for **GNU GPL v3** (copyleft). GPL would require derivatives to also be open-source, aligning with a *"share-alike"* philosophy. However, GPL might be too restrictive for some creative uses, so MIT (or Apache 2.0, which adds patent clarity) is likely preferred for maximal **community trust and adoption**.
- **Documentation and Whitepaper:** These written works can be licensed under a **Creative Commons** license. For example, **CC BY-SA 4.0 (Attribution-ShareAlike)** would allow others to remix or share the documentation as long as they credit the project and keep derivatives under the same terms. This matches the project's desire to spread knowledge while preserving the ideological integrity (people can't publish the docs without credit or under a closed license). CC BY-SA is akin to a GPL for content. If even more openness is desired, **CC BY 4.0** (Attribution only) or **CC0** (public domain) could be used, but given the careful curation of the project's narrative, BY-SA seems appropriate to ensure derivations remain in the same spirit.
- **Visual Assets & Media:** Any images, graphics, or aesthetic elements can also use Creative Commons licenses. If the assets are more art than utilitarian, the project might use **CC BY-NC-SA** (which adds Non-Commercial, to prevent commercial exploitation without permission). But NC can conflict with free use, so the team may avoid it to stay truly open. Likely they stick with CC BY or BY-SA here as well.
- **Name and Trademark:** The unique names (Kobalt, Ūmöwt, etc.) may function as trademarks or service marks of the project. The bundle could include a note that these names are part of the project's identity and should be used in context (to prevent confusion with other products). This isn't a license per se, but a clarification for intellectual property clarity [29] [30] – aligning with the notion of establishing *"solid legal and authoritative basis"* around these codenames. If domain names have been registered (like `kobalt.io` or others suggested in the docs [14]), those are listed for transparency.
- **Third-party Components:** The project likely includes or builds on some third-party libraries (e.g., maybe a physics math library, or dat.GUI if used). The bundle should include a **NOTICE** or acknowledgments for those, and ensure compatibility of their licenses with the chosen overall license. For instance, if any code was derived from GPL code, then the project would need to be GPL. Assuming all is original or MIT/Apache-type, there's no conflict.

Modular License Structure: In practice, the bundle might contain multiple license files or a section in the main LICENSE file delineating the parts: - `LICENSE` (main, for code), - `LICENSE-Docs.txt` (for documentation), - `LICENSE-Media.txt` (for assets), - Or a single file that states: "Code is MIT licensed, documentation is CC BY-SA, assets are CC BY," etc.

This modular approach ensures each aspect of the project is covered by an appropriate license, reflecting how the project spans multiple domains (software, text, art). It reinforces trust – users and contributors know exactly what they can do with each part. The open licenses also invite external collaboration, consistent with the project's community-oriented goals.

Conclusion

The assembled bundle delivers everything needed to experience and extend the **Interconnected Exosystème**. From a quickstart README to deep-diving whitepaper, from a live fractal network simulation to the configuration that ties it to real-world domains, it paints a complete picture of the project's scope. The engineering is **exhaustive yet elegant**: a groovy hum of code running just above the noise, inviting tinkering and discovery. The inclusion of archetypal narratives and multi-layered metaphors doesn't detract from the practicality – instead, it inspires a more mindful engagement with the technology.

In this package, **abstract materialism meets concrete implementation**. One can literally watch abstract ideas self-organize on screen, and then inspect an individual "electron" in the system through the UI. The bundle stands as a **bright, minimalist edition** of the project's ideology – everything in its right place, nothing superfluous, but with depth available on demand. By openly sharing this work (with permissive licenses and thorough documentation), the creators challenge existential assumptions about art and tech: showing that not only can they coexist, but they can enhance each other in a meaningful way.

Ultimately, this complete bundle is more than the sum of its parts (code, graphs, files) – it's a **meta-artifact** that embodies the very principles it espouses. In bridging disciplines and scaling from micro to macro, it offers a platform where users can explore **order from chaos** and **insight from interplay**. We leave the user with an invitation to dive in, tweak a parameter, follow a path from transistor-level to idea-level, and perhaps even contribute their own twist to this living system. *Let it be* – the Exosystème is ready to run. **Big kudos** to the journey ahead.

Sources:

- Project ideology and naming conventions [4] [5] [2]
- Whitepaper and domain ecosystem analysis [13] [3]
- Architecture overview and agent design [6] [7]
- Exotil 2D simulation code (network center of mass, clustering forces) [22] [23]
- Creative playground ethos (Spectra Gallery) [16] [31]

---

[1] [2] [3] [11] [12] [13] [16] [17] [18] [19] [21] [31] Spectra Gallery Domain Ecosystem – A Cross-Disciplinary Abstraction.pdf
file://file-LXzycAg18rk5MFpu5qgcyt

[4] [5] [8] [9] [10] [14] [15] [27] [28] [29] [30] Introduction.pdf
file://file-2pc4CXWUPjZ8pXPU38jXEz

[6] [7] [20] [25] [26] Spectra_Architecture_Overview.md
https://github.com/Exosysh-Lob-Ueht/arquolab-sandbox-models/blob/5d500468060e5965b4ee5f7750672fbf0b800ec8/docs/Spectra_Architecture_Overview.md

[22] [23] [24] exotil.html
file://file-MYFkwXUsUGxY1eeRfPHP7B