Deploying Artalytics Development Environment

This guides the creation of a deployment script allowing Codex, other Agents, and developers quickly launch a development and testing environment for any of the Artalytics Platform's R packages.

One motivation of this document is to achieve a setup script that can build the environment in under the timeout limit enforced by OpenAI's Codex agent.

Note on Codex Setup Script Timeout (1200s Limit)

OpenAI's Codex environment imposes a fixed time limit (around **10–20 minutes**) for the setup script to run, and this **timeout cannot be changed by the user**. In practice, environment provisioning scripts must complete within that window or the setup will be terminated. Therefore, our focus should be on **speeding up the installation process** rather than expecting to extend the time limit.

To confirm, OpenAI's Codex changelogs have indicated a ~10 minute default for setup scripts (with some users observing timeouts even earlier on certain plans). This limit is a platform constraint, so we should **assume ~1200 seconds is the maximum** and design our approach accordingly.

GitHub Remotes and R Package Compilation

Your current `codex-env-setup.sh` script installs a lot of system libraries and R packages (including **16 GitHub repositories**). The slowest steps are typically:

- **Installing numerous R packages from source** – especially those with compiled code or many dependencies. Even with `remotes::install_github`, these happen sequentially.
- **Downloading and compiling internal packages from GitHub** – network latency for multiple Git fetches and the compilation of each package can add up.

- **CRAN dependencies** – packages like **bs4Dash**, **shiny**, **magick**, etc., must be downloaded (and possibly compiled) if not available as binaries.

Given these bottlenecks, we need strategies to **reduce installation time**. Key approaches will include using more efficient installation tools (like `{pak}`) and leveraging caching or pre-built binaries.

Artalytics Remote Package Dependencies

Install pak if not already installed:

```
install.packages("pak", repos = "https://cloud.r-project.org")
```

Instead of multiple `remotes::install_github()` calls, use one `{pak}` call with the full list of required packages.

```r
pak::pkg_install(c(
  "RinteRface/shinyNextUI",
  "JohnCoene/waiter",
  "r-data-science/rdstools",
  "r-data-science/rpgconn",
  "artalytics/artcore",
  "artalytics/artutils",
  "artalytics/artbenchmark",
  "artalytics/artopenai",
  "artalytics/artopensea",
  "artalytics/pixelsense",
  "artalytics/artpipelines"
  # (Skipping Shiny modules and app for now to reduce load)
))
```

This single command will resolve all dependencies (including CRAN packages like **bs4Dash**, **shinyWidgets**, etc., needed by your packages) and install everything in an optimal order. The `{pak}` solver will ensure that, for example, **`artcore`** (the foundation) is installed before **`artutils`**, and so on, according to the dependency graph. It also means shared dependencies (like `rdstools` which is used by multiple packages) are downloaded once.

> **Why pak?** As an illustration, `{pak}` can install multiple packages in parallel, whereas `remotes::install_github()` handles them one by one. This parallelism and smarter dependency resolution make it *much faster* for large setups. The `{pak}` documentation notes it was created to **make package installation faster and more reliable**, performing all HTTP operations concurrently.

Caching and Binaries to Reduce Build Time

Beyond using `{pak}`, consider these tactics to further speed up installation:

- **Use Package Manager binaries:** Since base image is Ubuntu, you can point R to RStudio's Public Package Manager (PPM) for CRAN packages. The cloud CRAN mirror is fine (it often redirects to PPM automatically). This means many CRAN packages (and popular GitHub packages that PPM builds) could install as precompiled binaries. Fewer source builds = faster setup.
- **Leverage global cache (for dev, if reusing environment):** If you or other developers run the setup repeatedly on your own machine, look into `{pak}`'s caching or `renv` global cache. Both pak and renv store package builds in a global cache (`~/.cache/R/packagemanager` or `renv` cache) so that subsequent installations are much faster (they can use cached builds).

Using `renv` for Portable, Reproducible Environments

The `renv` package can manage project-specific libraries and lockfile snapshots of all dependencies. How this might look for Artalytics:

One unified environment: consider a top-level project (perhaps the main Shiny app repo, or a separate "dev environment" repo) that includes **all** Artalytics packages in its lockfile. For example, your `appPlatform` DESCRIPTION already references many internal packages. You could initialize `renv` in the appPlatform project after installing everything, and then snapshot. The lockfile will capture all Artalytics packages (with their GitHub refs) plus CRAN packages. This lockfile essentially becomes a definition of the entire platform's environment. Both Codex and human developers can use it to recreate the full stack in one command.

Implementing that now would make it **very easy to spin up a dev environment**: just clone the repo and run `renv::restore()` (which under the hood could use `{pak}` as the resolver for speed, or you can use pak directly on the lockfile with `pak::pkg_install("renv.lock")`). This approach ensures consistency between Codex, CI, and developer machines.

Pre-Built Docker Image (Optional)

For portability and deployment ease, another strategy is to distribute a **pre-built Docker image** containing all the necessary components.

Currently, Codex uses the `ghcr.io/openai/codex-universal:latest` base and doesn't support user-provided images (at least at the time of writing). But you could still use the Docker image for local development or Codespaces. For example:

- Push your pre-built image to a registry (GitHub Container Registry or Docker Hub).

- Developers can run `docker pull your-org/artalytics-dev:latest` and start an RStudio Server or VSCode dev container with all packages installed.
- This image could also serve as a caching layer for CI or Codex: even if Codex can't use it directly, you know exactly what needs to be installed since it mirrors the Docker build.

This is an optional path, but it aligns with the goal of a **highly portable, easy to deploy** environment.

Setting Environment Variables & Automating Config

It's crucial to configure the environment **correctly for the platform's needs**. In particular, the Artalytics R packages rely on several environment variables to know how to behave (especially for database and API connections). We must ensure these are set up in the Codex environment (and for developers).

Key environment variables include:

- **ART_RUN_AS_DEMO** – Should be `"TRUE"` or `"FALSE"` (as a string) depending on if you want the app to run in demo mode. Demo mode prevents any persistent modifications (no DB writes, no file uploads). For a Codex or test environment, you might set `ART_RUN_AS_DEMO=TRUE` to avoid accidental data changes.
- **ART_USE_PG_CONF** – Specifies which database config to use (e.g., `"artprod"` for production or `"artdev"` for a dev database). Your `artcore` package reads this to decide which DB connection settings to pull from the config file. In most cases, set `ART_USE_PG_CONF=artprod` if you want to connect to prod by default (or `artdev` if you have a dev DB).
- **ART_PG_USER_PASSWD_PRD / ART_PG_USER_PASSWD_DEV** – The actual database password for the `shiny` user in prod or dev, respectively. These are sensitive and should be provided securely (never hard-coded). In practice, you'd supply these as environment secrets. The `rpgconn` config will reference them via `!expr Sys.getenv("ART_PG_USER_PASSWD_PRD")` so that the password isn't stored in plaintext in the config file.
- **ART_BUCKETS_KEY_ID / ART_BUCKETS_KEY_SECRET** – Credentials for your DigitalOcean Spaces (for CDN asset storage). Needed if you want to use `artcore` functions that access the cloud storage. In a dev or Codex scenario, you might set these to dummy values or leave empty unless you are specifically testing those integrations.
- **ART_OPENAI_KEY, ART_OPENSEA_KEY** – API keys for OpenAI and OpenSea integrations (used by `artopenai` and `artopensea` packages). Again, provide the real keys if you intend to call those APIs, or use placeholders (like `"NO_VALUE"`) if not testing that functionality.

- ***R_CHECK_SYSTEM_CLOCK*** – You have this set to `0` in your snippet, which disables a check in R CMD check about system clock (helps avoid spurious warnings in containers). It's good to include for consistency.
- **CODECOV_TOKEN** – Used for test coverage reporting (covr -> Codecov) if you run coverage. Not needed for normal dev use unless you're uploading coverage results. In CI, it's provided as a secret.
- **GITHUB_PAT** – (Already handled) Required to install private GitHub packages non-interactively. For Codex, you must have this exported, as your script enforces. For devs, they should have a PAT in their `.Renviron` or environment as well, so that installing internal packages or running CI locally works. Remember not to commit this; use `.Renviron` or a secret store.

How to set these up? For a one-time Codex environment, you might directly export these variables at the start of the setup script (except secrets, which you'd ideally inject via the Codex UI or an env configuration). For local development, you can put them in `~/.Renviron`.

Automating `rpgconn` Database Config

One often-overlooked step is setting up the **PostgreSQL connection config** for the `rpgconn` package that `artcore` uses. Normally, after installing **rpgconn**, you'd run `rpgconn::edit_config()` interactively to create the config file. We want to automate this so the Codex environment (or a fresh dev machine) is DB-ready without manual editing.

The `rpgconn` default config path is `~/.rpgconn/config.yml`. We need to ensure this file exists with the correct content. Based on the platform's requirements, it should define an `artprod` connection that uses the `ART_PG_USER_PASSWD_PRD` env var. For example, the YAML should look like:

```yaml
config:
  artprod:
    host: "artalytics.app"
    port: 5432
    user: "shiny"
    password: !expr Sys.getenv("ART_PG_USER_PASSWD_PRD")
```

We can create this file in the setup script. For instance, at the end of the script (after installing rpgconn and artcore):

```bash
mkdir -p "$HOME/.rpgconn"
cat > "$HOME/.rpgconn/config.yml" <<EOF
config:
  artprod:
    host: "artalytics.app"
    port: 5432
    user: "shiny"
    password: !expr Sys.getenv("ART_PG_USER_PASSWD_PRD")
EOF
```

This writes the needed config. Notice the `!expr` syntax – this is exactly what `rpgconn::edit_config()` would insert. It means the password is pulled from the environment variable at runtime, keeping your secret out of the file. Once this file is in place, `artcore::..dbc()` (which reads the config) will find the `"artprod"` entry and use it, connecting with `host=artalytics.app` and the given credentials.

Alternatively, since you have `rpgconn` >= 0.2.0, there might be a helper function `rpgconn::use_config(path, overwrite=TRUE)` that can install a given config. But simply writing the file as above is straightforward and avoids any interactive steps. After creation, you can even do a quick check in R, e.g. `Rscript -e "rpgconn::config_paths()"` to verify it picks up the file (optional).

Double-check: Ensure `$HOME` is correctly set (in Codex it should be, and you have sudo so likely running as a default user). The `cat` approach is non-interactive and should work in the Codex setup.

Summary of Recommended Changes

To wrap up, here's a consolidated strategy:

- **Use `{pak}` for installation** – dramatically speed up package installs by doing them in one go, in parallel. This addresses the timeout issue by shortening install time.
- **Install only what's needed initially** – Skip Shiny module packages and the app if Codex doesn't require them immediately, to stay well under 1200s.
- **Consider `renv` for dev environments** – implement renv lockfiles for reproducibility. This will make it easy for any developer (or CI, or even Codex with the lockfile) to recreate the environment reliably with `renv::restore()` or `pak::pkg_install("renv.lock")`.
- **Prepare environment variables** – Load all required env vars (database config selector, demo flag, API keys, etc.) via a `.Renviron` or through exports. This ensures the packages behave correctly (e.g., using the production DB config but perhaps running in demo mode to avoid writes). Don't forget to provide `GITHUB_PAT` (for installs) and any needed keys securely.
- **Automate DB config setup** – Add steps in the script to create the `~/.rpgconn/config.yml` with the `artprod` settings pointing to your production DB and using the env var for password. This saves manual effort and allows Codex or tests to call `artcore::..dbc()` without error.

- **Test in the Codex base image** – Once you implement these changes, you should test the script in an environment based on `ghcr.io/openai/codex-universal:latest` (which is the base). You can simulate this by running that Docker image locally. Verify that the entire script completes under 20 minutes. Given the improvements, it likely will complete in perhaps ~5-10 minutes (depending on network and build speeds). All packages should install and you should see the " CODEX environment setup complete!" message.
- **Future: pre-built image** – Optionally, maintain a Docker image with everything pre-installed. This can serve as a quick-start for new developers (just run it and everything is there), and if OpenAI ever allows a custom image for Codex, you could use it to avoid the setup phase entirely.

By following these steps, you'll achieve a **fast, reproducible, and portable development environment** for the Artalytics platform. Codex will be able to set up the core packages quickly, and human developers (including you) can easily launch a full environment for any package or for the whole ecosystem with minimal fuss. This strikes a balance between **speed** (using pak and caching) and **maintainability** (using renv and proper configuration management). Good luck, and happy coding!

Sources:

- Artalytics AGENTS Guidelines (env. variables & reproducibility)
- Internal Dockerfile (installation & config example)
- R `{pak}` Documentation (fast parallel installs)
- Dataquest Tutorial on `{pak}` (installation speed)
- Artalytics CI Notes (using PAT and secrets in env)