Anthony Sherrell

Otago ID: 6555287

COSC349 – Cloud Computing Architecture

8 September 2025

Assignment 1

LiliumShare is a small, Parsec-like screen-share tool built to show how to build and deploy in a portable way to three cooperating VMs. The application intentionally partitions work amongst a PostgreSQL VM for storage, a Node.js API VM for business logic and signaling, and a nginx "edge" VM which fronts the API and provides a single point of entering for clients. A python-based GUI runs on the host.

The three-VM topology is rather simple and straightforward. The VMs are:

1. The database VM ("vm-db") which installs Docker inside the guest and runs a single postgres:16 container using a persistent Docker volume;
2. The API VM ("vm-api") which also installs Docker, builds a local image for the node backend from the repository's Dockerfile, waits during provisioning for the database VM to accept TCP connections, runs a one-shot migration container to apply idempotent schema DDL, and starts the backend container that serves both the REST and a WebSocket endpoint
3. The edge VM ("vm-edge") which installs nginx and publishes one server block which proxies both HTTP and "/ws" requests to the API VM.

All three VMs share a host-only private network and the NAT forward can expose the edge on 127.0.0.1 when the host port is available. The python GUI uses a single JSON file that includes addresses, making it easy to avoid magic number madness (a major problem at the beginning of the project).

For clarity, isolation, and relevance to the assessment, I would opt for three separate VMs over a single server. First, the use of three VMs is representative of how most cloud deployments get decomposed. The large object storage has a different lifecycle/security boundary from the API (which is also stoppable/rollable), and the edge tier can address cross-cutting concerns like TLS, or WebSocket upgrades. Second, in separation, faults are more graceful - if the API container crashes, the database VM still has state, and liveness probes are still functioning; an nginx configuration error means I can check the api directly from vm-api. Finally, decomposition allows for a clearer demonstration of reproducibility - the provisioner for each VM is both idempotent and short (timeouts); failures can only take a single area with separate (human) terminations; and the log files are much more readable.

All ports and networking configurations are centralized in backend/networkconfig.vagrant.json, with LILIUMNETCFG present for any override options. The Vagrantfile will not allow a boot if there is any missing keys, all required keys are checked - we have no magic numbers hard-coded into our provisioners from what I noticed. The edge's NAT forward is even more complex: the Vagrantfile first checks if there is an available host port before adding a forward. If the port is used, the VM will omit the forward entirely and a post-up message will print the host-only URL to use. This small guard opened the door to remove an entire class of confusing failures from earlier prototypes where VirtualBox would re-direct to a different host port that was also unexpected and I would be testing a different address without knowing it.

I have two entry-points into the automated build. For the majority of people, buildproject.sh is the easiest way: mark it as executable and run it from the root of the repo. It will vagrant-up all three VMs, build the backend image on vm-api, ensure the database is up on vm-db, run migrations, setup nginx on vm-edge, wait for /health, and (optionally) bootstrap three demo users so that the GUI has interesting state when it runs for the first time (see the README for more details). For iteration on the virtualized stack itself, scripts/rebuild_from_scratch_vagrant.sh offers a more explicit flow with flags like --no-cache (which forces a cold Docker build).

The primary software that the build will download is predictable. A clean first time build will pull per-VM APT packages for Docker and nginx (tens of megabytes), along with Docker base images for postgres:16 and node:20-slim (hundreds of megabytes combined). In my measurements, the first build pulled about 15 MB outside of container layers, and once the first run was complete (including the auto-generated demo users), the repository directory had grown to about 495 MB. The three VirtualBox guests took up about 11.9 GB on disk after provisioning (vm-db ~4.3 GB, vm-api ~4.1 GB, vm-edge ~3.5 GB). Any subsequent redeployments are not going to download a ton of new data; only upgraded OS packages and changed application layers will be downloaded, as the rest of the layers are cached in the guests. Resource allocations are fairly modest: vm-db and vm-edge are set to 1 vCPU/1 GiB, and vm-api is set to 2 vCPU/1 GiB. These allocations are documented in the README file and can always be modified in the Vagrantfile if a developer wants to set it up differently.

For a developer to access the application, they need to run the build script (making it executable) and, on Wayland desktops, allow the portal's first run consent when the GUI prompts to capture the screen. Otherwise the system wants to be unattended: Vagrant provisions each VM idempotently, migration can be run one-time and is safe to apply again, nginx installs and reloads without prompts, and the scripts block on health checks passing. If the edge's NAT forward is skipped because the host port is busy, the scripts print and automatically reuse the host-only URL to avoid confusion. Please note that the app does NOT currently work on Wayland desktops for some unknown reason, the 'consent' window does not pop up when it should. Testing has not been done on an X11 environment, but it should theoretically work there.

Because the assignment is fundamentally about being transparent around the process of debugging, I've outlined the major issues I faced and how I went about fixing them:

1. The very first call to provision vm-api, it tried to install the docker-compose-plugin package, and failed with unable to locate package docker-compose-plugin. The fix was to fold the VM toolchain into a simpler call. Rather than needing Compose inside the VM, I made the provisioners into standard docker build and docker run for the backend, while leaving Compose for optional single-host development on my machine. This eliminated a redundant dependency and the API VM provision works consistently on unspoiled Ubuntu 22.04 boxes.
2. The early migrations were failing with Error: connect ECONNREFUSED 127.0.0.1:5432 although the database VM was running. The root problem was that the migration container defaults to DATABASEURL or localhost and ignored the netcfg VM-to-VM IP. I fixed it two ways. First, migrate.js will look for LILIUMNETCFG if

DATABASEURL is not set. Second, the provisioner calls docker run with explicit DBHOST DBPORT DBUSER DBPASSWORD and DBNAME environment variables taken from the json. After these two fixes the migration logs included Migration completed and the API container started reliably on the correct port.

3. The vm-edge timed out during boot sometimes, with Vagrant logs indicating “Fixed port collision for 18080 …” and then a long wait and fail. The issue was port-forward churn: VirtualBox would auto-correct to a different host port, while my scripts still assumed 127.0.0.1:18080. nginx looked healthy, but I was curling the wrong address. The sustainable fix was a small Ruby helper, portfree?, in the Vagrantfile to only add a NAT forward if the requester-port was actually available. If it was not available, it skips the forward on purpose, and the post-up message tells you to use the host-only URL instead. I also raised the edge VM's boottimeout and memory a little, to try to rule out slow boots as a cause. Since then, vm-edge has been working reliably.
4. I briefly broke the rebuild script by issuing docker commands on "vagrant ssh -c ..." without sudo, leading to nothing being done on the host and Vagrant failing with "permission denied while trying to connect to the Docker daemon socket." Provisioners run as root; ad-hoc commands do not. I limited all container lifecycle to the provisioners (who run as root), and where I really wanted a remote call during a rebuild, I prefixed with sudo and passed through the relevant environment variables instead. The final vagrant oriented rebuild script is within these constraints and has not run into Docker permission issues again.
5. After switching over to the three-VM topology, the bootstrap script for some iterations often saw "502 Bad Gateway" even when curl /health was succeeding just moments earlier. I was still pointing the script at 127.0.0.1:18080 when the NAT forward had been

intentionally skipped. The solution was to have the script read the same JSON as the Vagrantfile, compute the base URL correctly (it's host-only when the forward is skipped), and then have it echo that before any HTTP requests. I also added a short back-off and a second health poll to let nginx reload after the API container is restarted.

6. On the GUI side, Wayland capture was not reliable on my machine. Rather than require system-level changes, I added a synthetic "screensaver" frame generator and two small sanity scripts: one checks for headless capture conditions, and one checks whether I can create OpenCV/pygame windows. The GUI now falls back to the synthetic pattern, and the console prints are clear, so I can still demonstrate the end-to-end signaling path even when real capture is not available on the host.

Two concrete classes of source changes — and how to rebuild — are worth calling out. The first is an API feature change. If you wanted to change something like add simple messaging between friends, would edit backend/src/server.js to add REST endpoints, and add a small messages table in migrate.js. After changing these two files, then you would run scripts/rebuild_from_scratch_vagrant.sh --no-cache so that it would force a complete rebuild of the backend image and reapplied the migration; the script waits for the /health to respond before returning. If you only changed server.js and not the schema, vagrant provision vm-api is good enough and is faster. The second case is a media pipeline change on the GUI; for example, you could be replacing the synthetic "screensaver" fallback with a PipeWire capture source that works on your desktop. Those changes are in frontend/screen_capture.py and the rtchost/viewer modules. After you have edited them, just recreate or activate the Python venv (source venv/bin/activate), run pip install -r frontend/requirements.txt if you know the dependencies changed, then just rerun the GUI processes with the right HOME directories; no need on the GUI for VM build changes.

The repository is structured for straightforward onboarding. The README explains the VM roles, the build and rebuild paths, and the networking model; it also includes targeted debugging recipes (logs, health checks, and common pitfalls such as 502s caused by a skipped forward). The commit history shows the evolution from a single-host Docker Compose prototype to a VM-based stack, the move to centralized configuration to eliminate magic numbers, hardening of migration and startup sequencing, and the final polish of the developer experience with one-touch scripts and deterministic network behavior.

Lastly, attribution: I utilized open-source software including VirtualBox, Vagrant, Ubuntu, Docker, PostgreSQL, nginx, Node.js, and some Python packages (aiortc, pygame, requests, and cryptography, and their transitive dependencies). I utilized ChatGPT during development to learn how to use certain packages, locate specific error messages (e.g. the ECONNREFUSED during migration and the nginx 502 for skipping the forward), and write documentation. All project code itself is mine. I also utilized Grammarly.com to help to fix grammar mistakes and suggest improved wording in this paper.