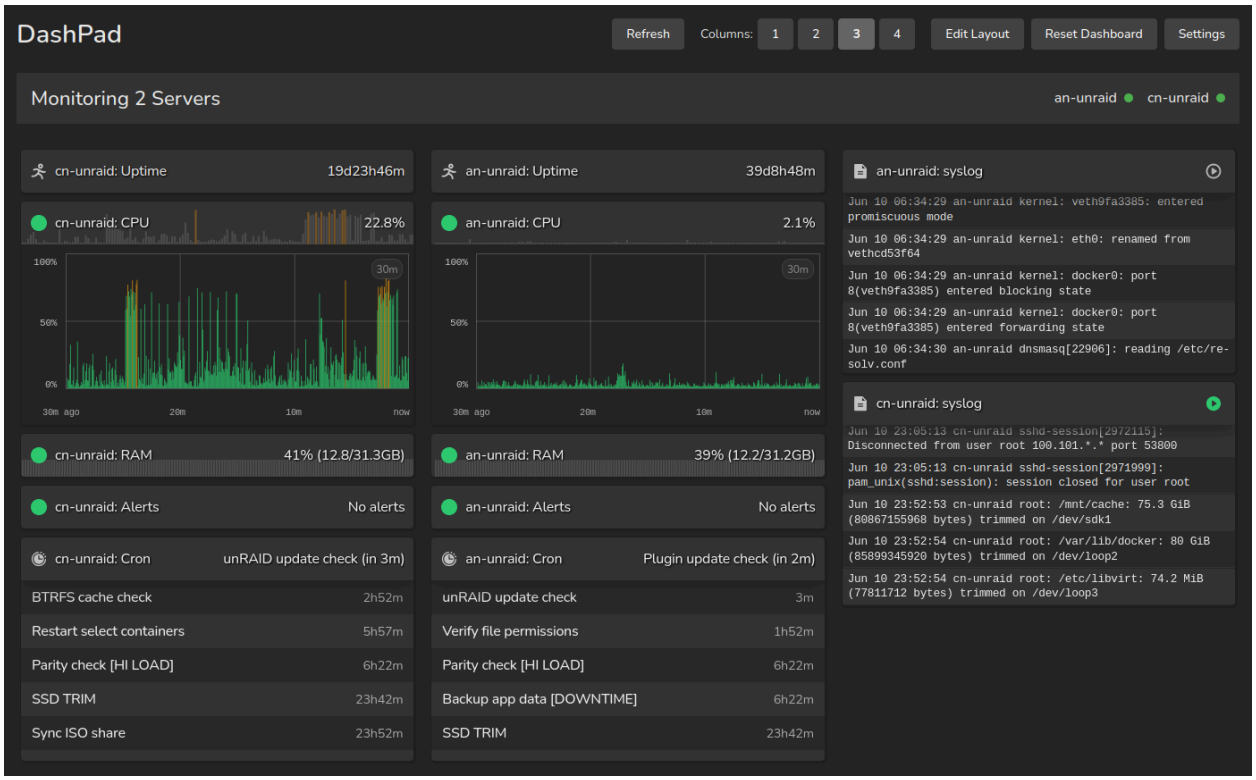
DashPad: Containerized System Monitoring For Tablets
June 19th, 2025
Chris Neal



Screenshot of DashPad-Web

Project Overview

System administrators and home lab enthusiasts need efficient infrastructure monitoring but often have limited budgets and unused tablets. DashPad addresses these challenges by repurposing old tablets as dedicated monitors, providing a singular interface for tracking multiple Linux-based servers simultaneously.

DashPad was built with a two-component microservices architecture, consisting of "DashPad-API" (Application Programming Interface; a FastAPI/Python backend container that collects system metrics, logs, cron tasks) and "DashPad-Web" (a Svelte/JavaScript frontend container optimized for tablet-based visualization and operation). The platform-agnostic design runs anywhere containers are supported, with the Web container deployed on Google Cloud Run successfully. Live system information updates every 2-60+ seconds (user-selectable), giving administrators flexible and efficient access to current infrastructure health data.

Project Stakeholders

Primary stakeholders include the project creator experiencing this exact problem, and system administrators needing near-real-time server insights. Other primary stakeholders include home lab enthusiasts running container-compatible platforms (particularly unRAID, a network data storage-focused operating system), and tablet owners seeking to extend hardware lifespan while reducing e-waste. Secondary stakeholders include the course instructor, possible future codebase contributors, and organizations who may wish to deploy DashPad for minimal, highly efficient monitoring.

Project Objectives

The main goal was to deploy a straightforward, functional monitoring dashboard on Google Cloud Platform at the smallest possible cost, demonstrating cloud services and containerization understanding. Technical objectives included multi-server monitoring with simultaneous display, 2+ second refresh rates with minimal resource usage, a tablet-optimized responsive interface, and HTTPS (Hypertext Transfer Protocol Secure; encrypted web protocol) communication. The architecture supports optional Netdata integration with fallback capabilities while maintaining operational efficiency.

Resources and Technical Architecture

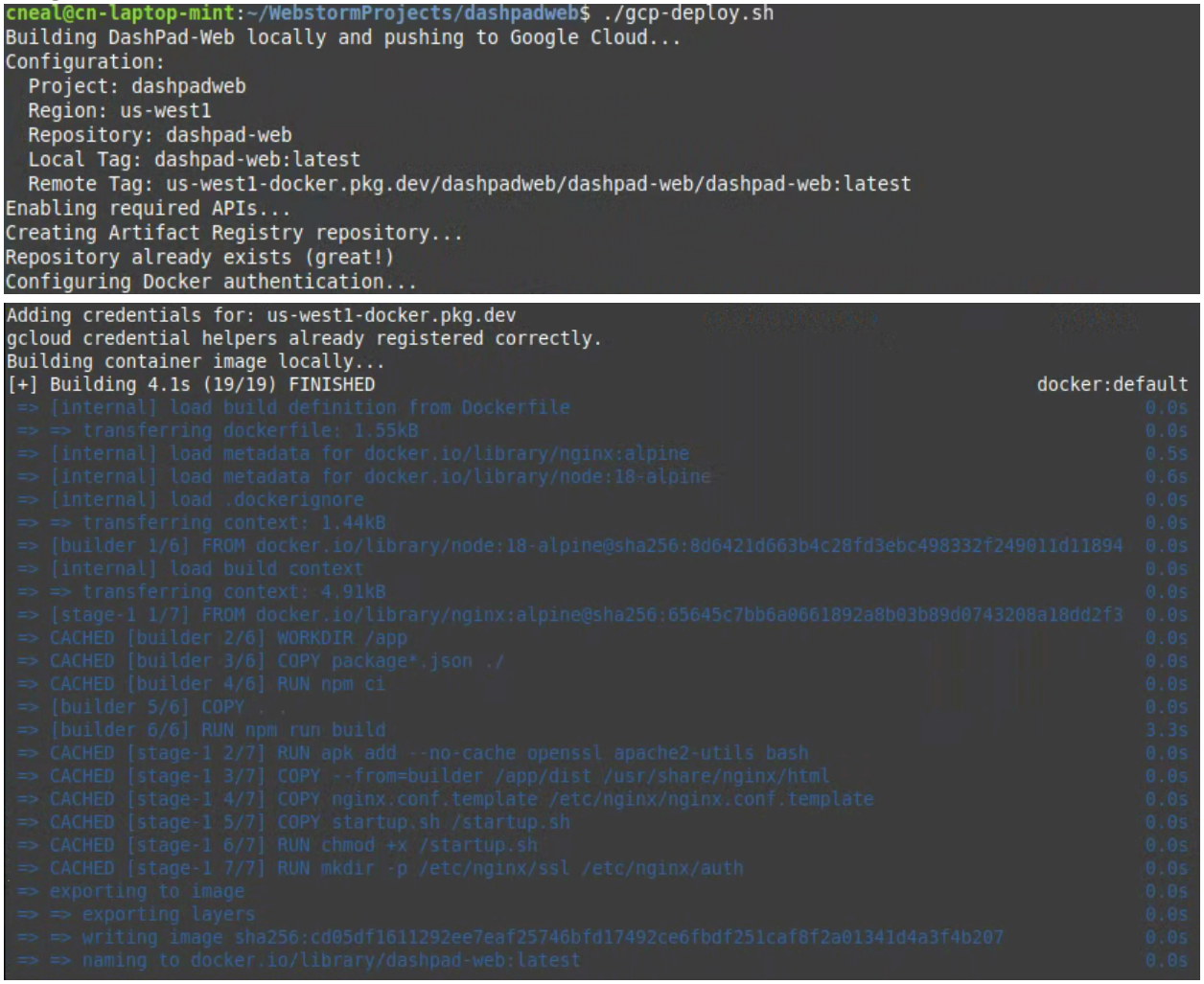
The project has evolved from early 2024 prototypes into a sophisticated, cloud-compatible solution. These prototypes leveraged a basic FastAPI implementation with HTML (HyperText Markup Language) for web page structure, CSS (Cascading Style Sheets) for webpage styling, and JavaScript to facilitate polling data from FastAPI. Initial plans involved single Docker containers with manual deployment and no security measures.

The current implementation leverages Google Cloud Run with serverless hosting, Artifact Registry for versioning, and Google-managed SSL (Secure Sockets Layer; used to secure web communications). The API container uses FastAPI with Python “slim” containers. It supports both direct filesystem reading and basic Netdata API integration. The Web container uses Svelte (chosen for compiled JavaScript efficiency and modular architecture) supported by an NGINX (pronounced “Engine X”; a web server) proxy. Configuration differs between components: JSON (JavaScript Object Notation; structured text data) files are used to configure the complex API container, and environment variables are used for the simpler Web container.

Key features include multi-server monitoring, SVG-based (Scalable Vector Graphics) charting, background sparklines, and a comprehensive debug panel. A unique “hinting” system provides timing hints in API responses for frontend synchronization. All data storage occurs in-memory on the viewing device. Preferences can be exported and imported as JSON.

Deployment Screenshots and Artifacts

Image 1 *- Build script output that ships the built image to Artifact Registry (series of 3 images):*



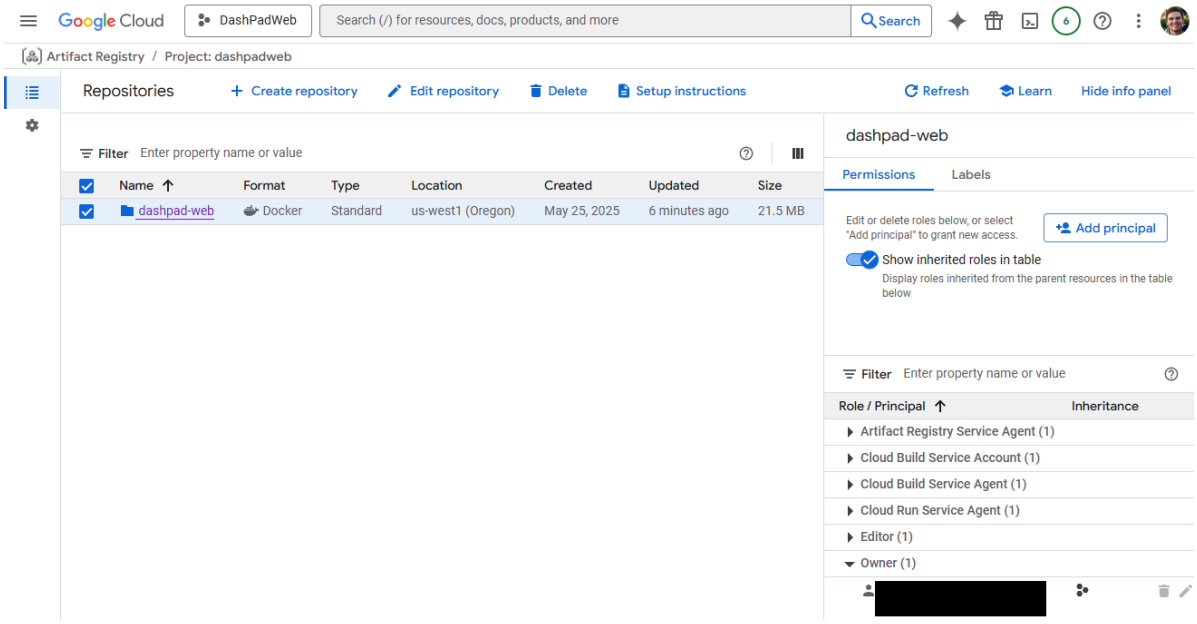
```
Tagging image for Artifact Registry...
Pushing to Artifact Registry...
The push refers to repository [us-west1-docker.pkg.dev/dashpadweb/dashpad-web/dashpad-web]
fa82c5c23104: Layer already exists
970b76e5b4a7: Layer already exists
49d49ffe6cab: Layer already exists
30679e4691a4: Layer already exists
c39be5debfb7: Layer already exists
90241f4aaedf: Layer already exists
0d853d50b128: Layer already exists
947e805a4ac7: Layer already exists
811a4dbbf4a5: Layer already exists
b8d7d1d22634: Layer already exists
e244aa659f61: Layer already exists
c56f134d3805: Layer already exists
d71eae0084c1: Layer already exists
08000c18d16d: Layer already exists
latest: digest: sha256:28229fa89e279f9769245b6411a6f05bb2cf9908613de02ecb84e3a243b8de7a size: 3241
Cleaning up local images...
Untagged: dashpad-web:latest
Kept remote-tagged image: us-west1-docker.pkg.dev/dashpadweb/dashpad-web/dashpad-web:latest
Build and push complete!

Image is stored at:
  us-west1-docker.pkg.dev/dashpadweb/dashpad-web/dashpad-web:latest

Image size information:
REPOSITORY                                                      TAG       SIZE
us-west1-docker.pkg.dev/dashpadweb/dashpad-web/dashpad-web   latest    52.6MB
cneal@cn-laptop-mint:~/WebstormProjects/dashpadweb$
```

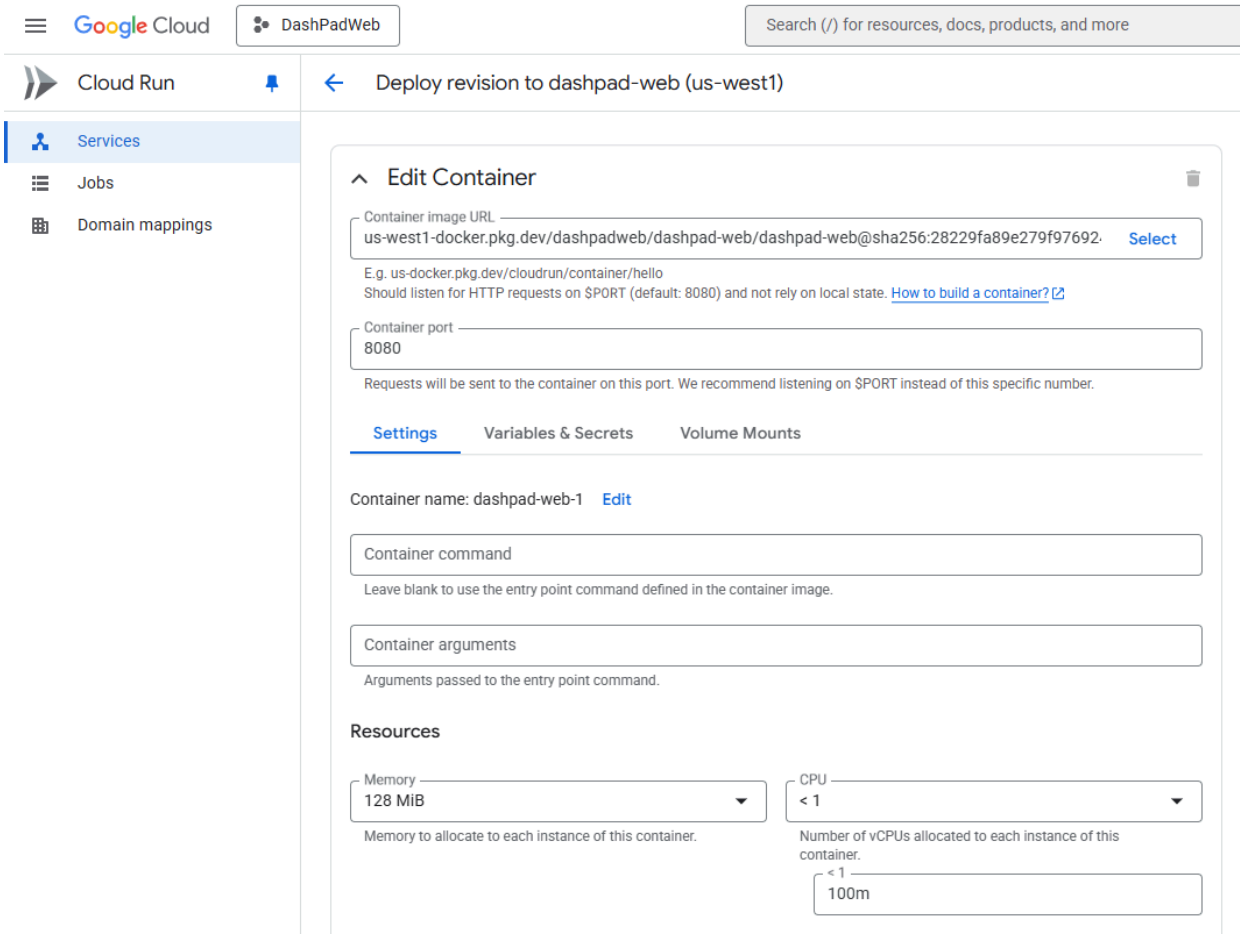
The script automates part of the deployment process for DashPad-Web. As shown in the output, the script builds a Docker container locally from the project's Dockerfile, then "pushes" it to Artifact Registry. The script checks for and enables Google Cloud Platform APIs, creates the repository if needed, authenticates with Google Cloud Platform, builds & tags the created image, then finally pushes it to the registry. More information regarding Dockerfiles can be found in a previous development post created by the author in July 2024.

***Image 2** - DashPad-Web in Artifact Registry, showing proof of size and account ownership:*



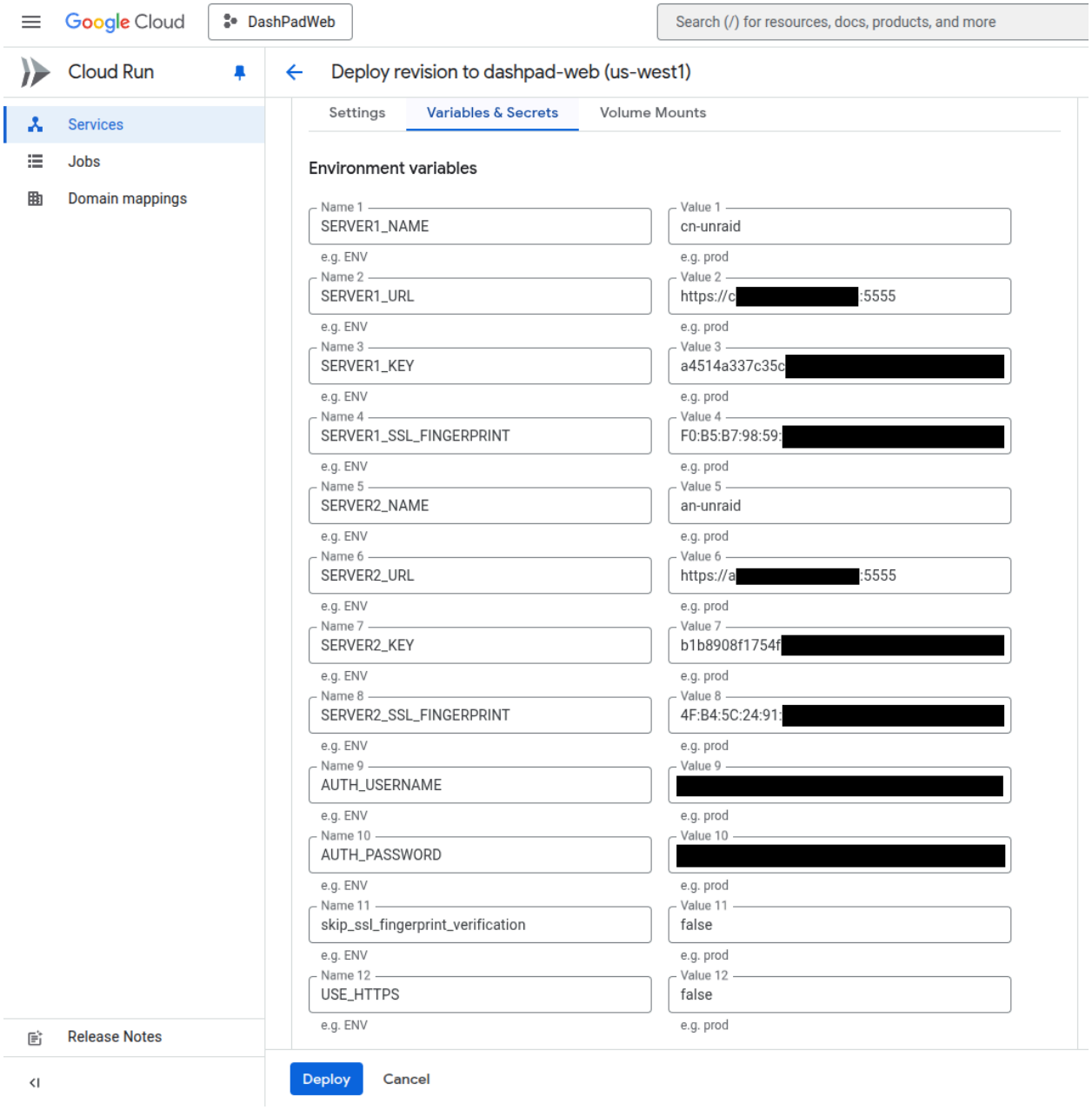
The size disparity between Artifact Registry and the terminal output is the result of image layer data deduplication. Approximately 30MB of the original image layers are already present in the registry from other images, resulting in a 21.5MB "virtual size." More on Docker image layers can be found in an exploratory development post created by the author from 2024.

***Image 3** - DashPad-Web container being configured with 128 MiB of memory and 0.1 vCPUs (a "Virtual Central Processing Unit" can represent fractions of a CPU, often used within cloud and virtualization environments); the container URL points to the image in Artifact Registry:*



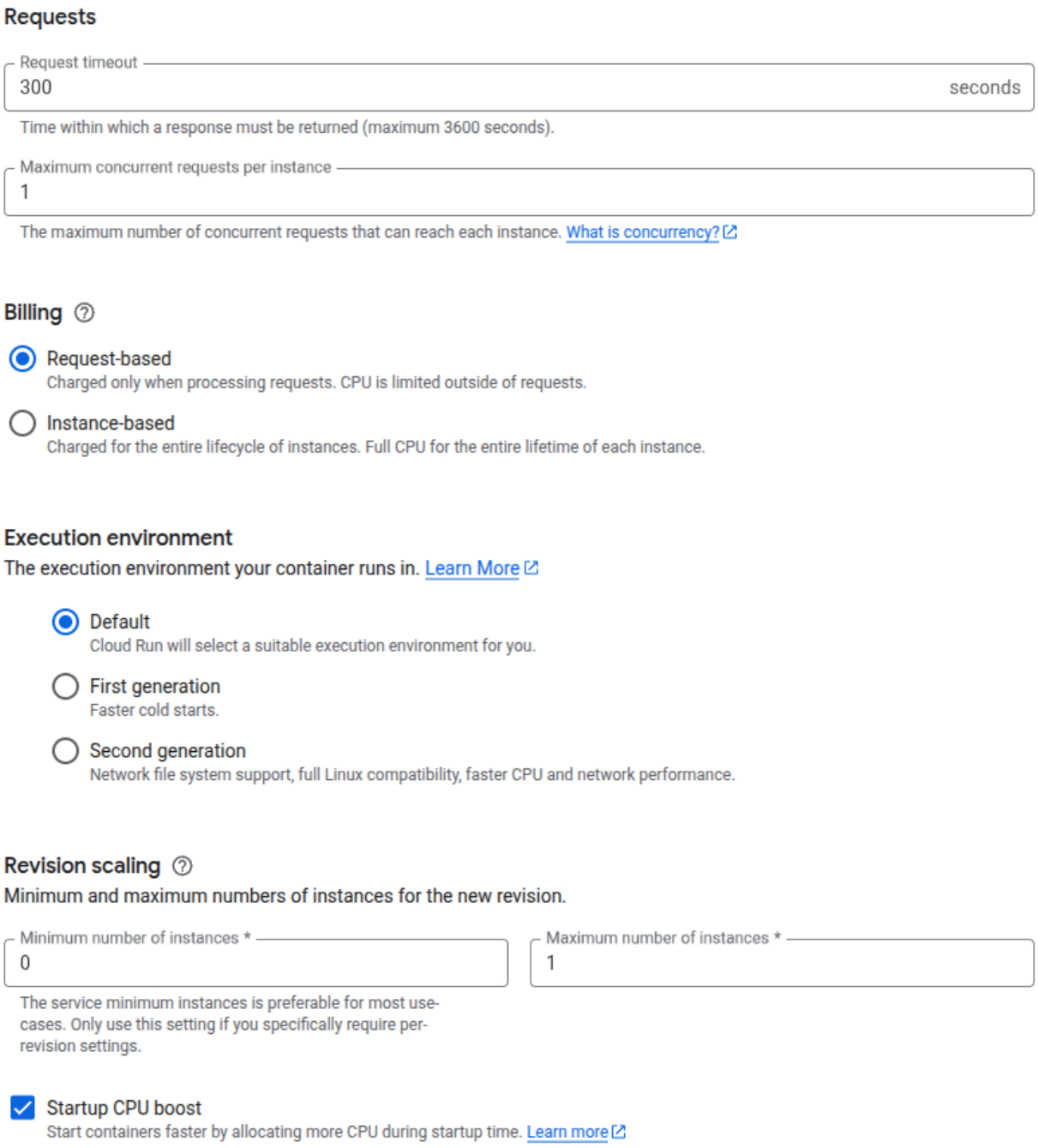
NOTE: As mentioned in the description under the "Container Image URL" field, the $PORT variable is injected into the container in single-container deployments. The Web container is configured to receive and listen on whatever port is provided via the $PORT variable, which Cloud Run provides for each container.

Image 4 *- The DashPad-Web container being configured using environment variables, including the name, URL, key, and SSL fingerprint of the API containers installed on monitored servers:*



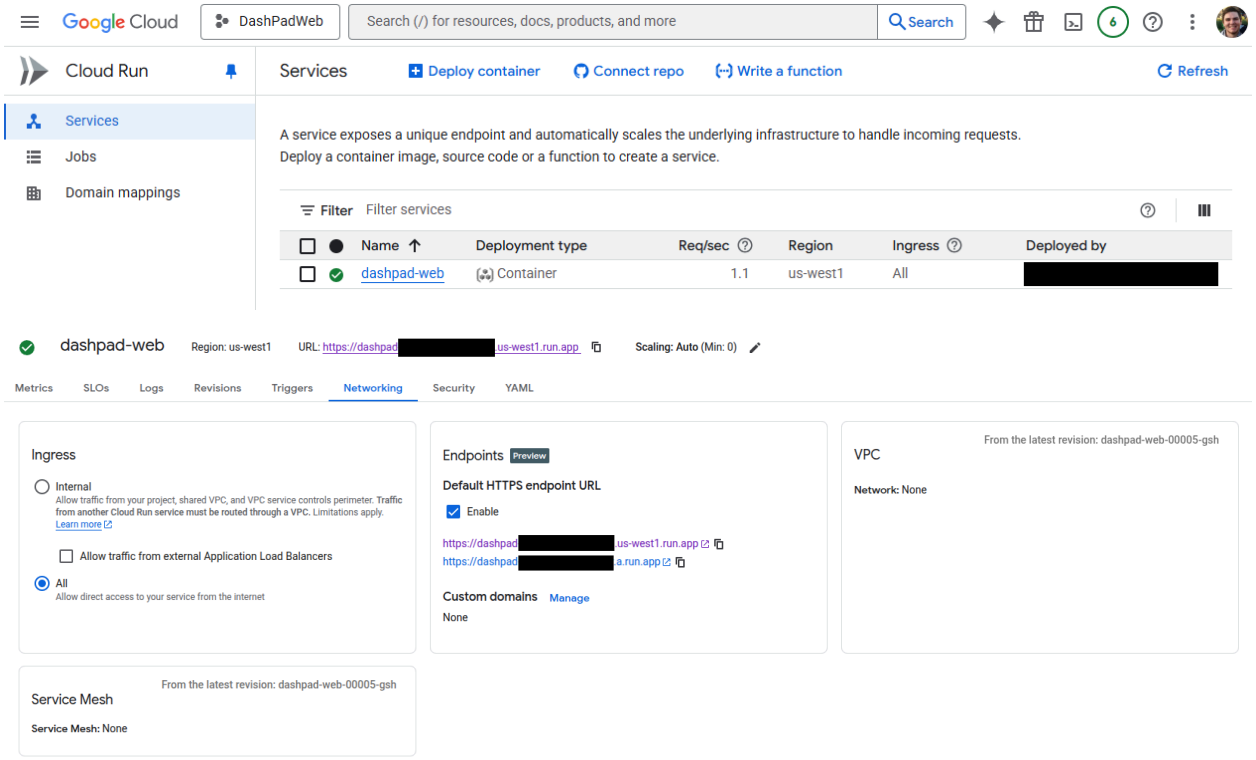
NOTE: It is **not** best practice to store secrets like passwords or other credentials in environment variables. The primary purpose of this screenshot is to illustrate how straightforward the configuration process is for the Web container.

Image 5 *- The DashPad-Web container being configured in Cloud Run, with "Request-based" billing selected. Maximum concurrent requests per instance are set to 1, and scaling is set to 0 and 1 instance (minimum and maximum, respectively), facilitating entirely on-demand billing:*



This means no costs are incurred at all if the Web application is not actively being accessed or viewed through a browser.

Images 6 & 7 *- The DashPad-Web container deployed and actively running within Cloud Run, displaying the full application URL assigned by Google with network ingress settings:*



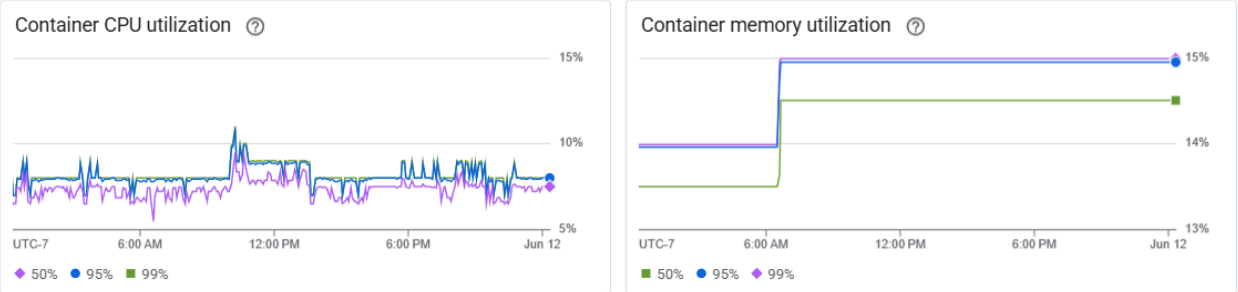
NOTE: While ingress traffic is set to “All”, the Web container ships with a simple, HTTP-based authentication system through its bundled NGINX instance. These credentials are visible in the environment variables shown within Image 5.

Technical Architecture

DashPad-API serves as the data collection engine, maintaining approximately 50MB (within 10MB of variance) of memory usage even after 24+ hours. It handles all processing logic, threshold determination, and provides dynamic update intervals to the Web container through a hinting system. Each API response includes a field indicating (in milliseconds) when the next data will be ready. The container is platform-agnostic to support deployment on any system, though direct data collection methods are Linux-specific. API keys, self-signed SSL certificates (to encrypt web-based traffic), certificate fingerprints, and configuration defaults are automatically generated and displayed (where applicable) in the container’s log file, facilitating ease of use with DashPad-Web deployment.

The DashPad-Web container operates as designed on just 0.1 vCPU (with 8% average usage) and using only 15% of the minimum-selectable 128 MB memory allocation over 24-hour testing (approximately 19.2 MB usage). The responsive 1-4 column layout with draggable and expandable modules provides tablet-optimized interaction, and local browser storage maintains user preferences and persistence. If the Web container is deployed in an environment that does not support automatic HTTPS communications (perhaps hosting on-prem), this container also generates and uses a self-signed SSL certificate for secure web communications (unless the USE_HTTPS environment variable is set to false).

Image 8 *- DashPad-Web CPU and memory statistics after 24 hours (8% usage on 0.1 vCPUs, 15% used of 128 MiB memory available):*



Implementation and Deployment

Development evolved through multiple phases, technically spanning over a year. “Phase 0” (March 2024 - March 2025) focused on experimentation, research, and technology exploration.

Phase 1 (April 14 - May 4, 2025) focused on research and technology selection. FastAPI was chosen for its proven ability to reliably serve data over an API, while Svelte emerged as the optimal frontend framework after comparative research. The microservices architecture was selected to enable future expansion, supporting additional display containers like a Discord bot (prototyped in 2024).

Phase 2 (May 5 - May 24, 2025) encompassed highly active development. JSON configuration was implemented for the API container’s numerous options, while environment variables simplified Web container setup. New security features include 64-character API keys and optional SSL fingerprint verification. Data censoring capabilities, including email addresses, IPs, URLs, and custom text from log modules, are also configurable. Build automation scripts reduced deployment time from 5-10 minutes to under 2 minutes. Most development time addressed bug fixes and stability improvements, reflecting the complexity of coordinating multiple containers.

Phase 3 (May 25 - Present) was characterized by successful Google Cloud deployment and ongoing refinement. The Web container became fully functional on Cloud Run by May 24, 2025. Deployment involved enabling required APIs (from the Google Cloud SDK or “software development kit”), configuring authentication, building containers locally, and pushing to Artifact Registry. Cloud Run configuration specified minimal resources (0.1 vCPU, 128 MB memory) with environment variables for multi-server support and 0-1 instance auto-scaling for pure pay-per-use billing.

Original timelines planned priority integrations by week 8. These include Jellyfin, Network UPS Tools, and unRAID-specific information. However, further refinement of the Web container’s module system is needed before implementation. Current efforts focus on developing comprehensive documentation to assist future users with setup and deployment. GitHub repositories containing both containers and other resources will be released upon completion of the documentation and project.

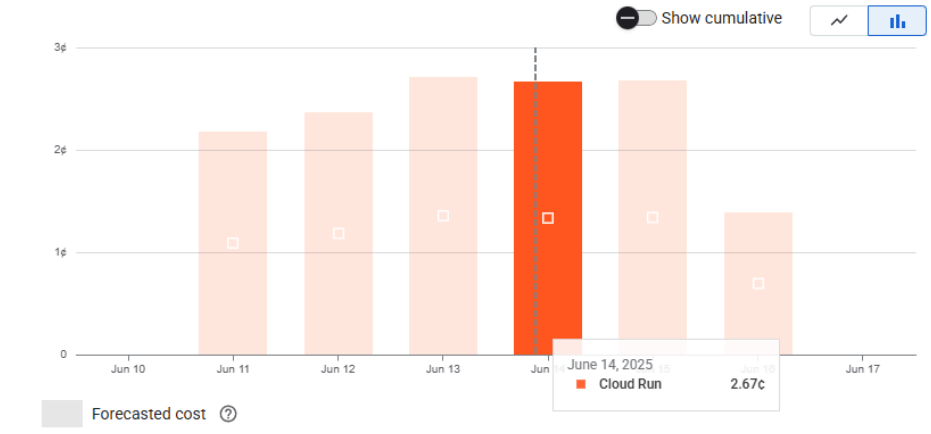
Results and Performance

All core objectives were successfully achieved. DashPad as a system does provide near-real-time updates every 2-60 seconds across multiple servers (where API containers are deployed) simultaneously. Cloud deployment on Google Cloud Run operates successfully with request-based billing, scaling to zero when idle. Performance metrics show API response times under 200ms and a 100% request success rate.

Image 9 *- Active API container deployment on an unRAID server, detailing two hundredths of a percent of CPU usage and 48.5 MB of memory used after 48 hours:*

Cost analysis demonstrates exceptional efficiency. After 24 hours of continuous operation (monitoring *two* servers using default values of 4-second metric intervals), DashPad-Web incurred just 2.67 *cents* per day ($0.0267). Using the 1.1 requests per second average measured via Cloud Run's Metrics page (with 0.1 vCPUs), approximately 2.85 million requests will be made per month. This yields a projection of $0.36 for Cloud Run compute. Since GCP's request-based billing model means cost scales directly with request frequency, end users can fine-tune their expenses by adjusting refresh intervals: Faster intervals increase costs while slower intervals reduce them. Artifact registry storage adds $0.03, assuming 100 MB for three DashPad-Web builds. Overall projected monthly costs are between $0.39 (calculated) to $0.80 (measured) based on actual usage patterns. Cost projects to roughly $2.40 at 3 months, $4.80 at 6 months, and $9.61 **annually** for customizable, flexible system monitoring at minimal cost.

Image 10 *- Cloud Run Billing Report after multiple 24-hour stress tests, showing a total average cost of $0.0267 (2 cents) over the span of one day (DashPad-Web container):*

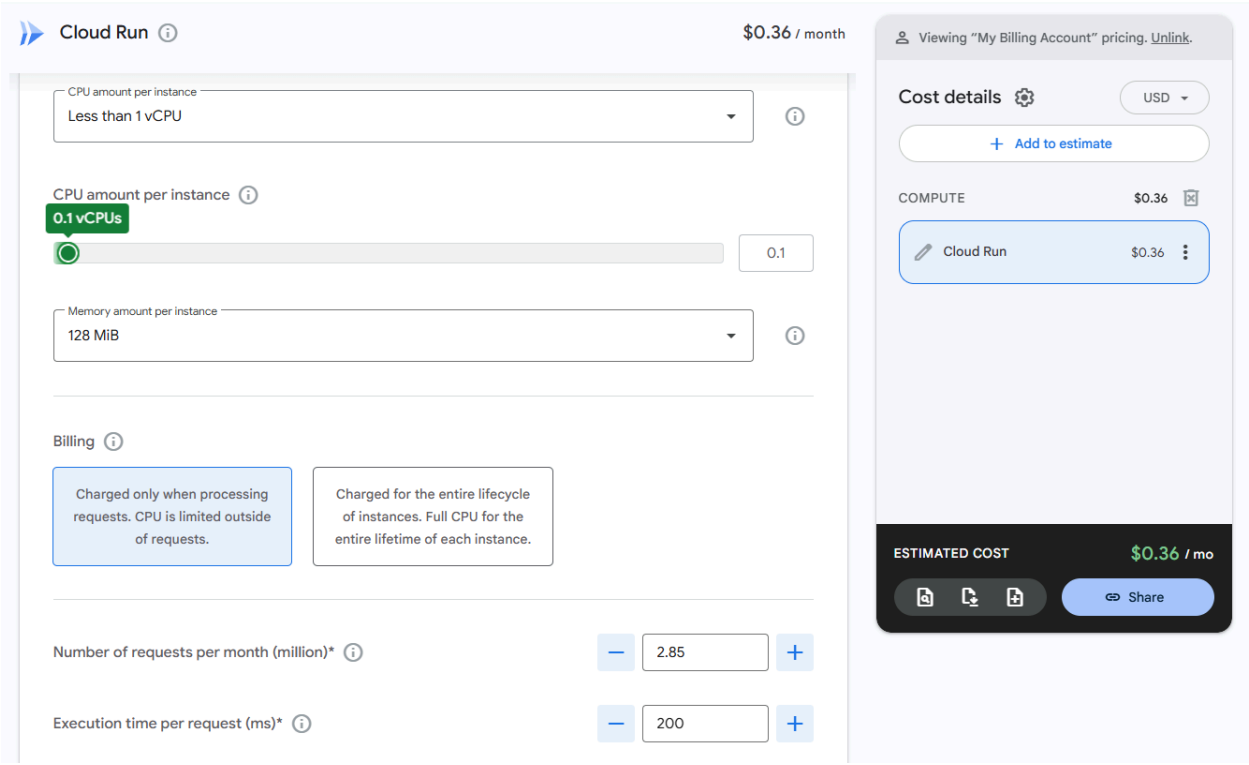


Download CSV

Service	Cost	Discounts	Promotions & others	Subtotal	% Change
Cloud Run	$0.14	–	–	$0.14	New
Artifact Registry	$0.00	–	–	$0.00	–

Subtotal	$0.14
Tax	–
Filtered total	$0.14

Image 11 *- Cloud Run calculator showing a projected monthly Cloud Run cost of $0.36 for a DashPad-Web container (excluding Artifact Registry costs, which adds $0.03):*



Challenges and Solutions

SSL certificate verification and API authentication presented the primary challenges. Supporting self-signed certificates for local (on-premises) deployments while maintaining security required implementing optional SSL fingerprint verification. Additional security features include data censoring in log modules, including IP addresses, email addresses, URLs, custom text strings, and MAC addresses (Media Access Control addresses are unique to individual network interfaces), plus censored server addresses in the frontend.

Frontend-backend synchronization initially encountered polling drift (timing desynchronization) and missed updates. The hinting system solution has each API response include "update_interval_sec" fields for each data type from each server, allowing the frontend to automatically adapt to API container schedules. This eliminated timing bugs while improving efficiency.

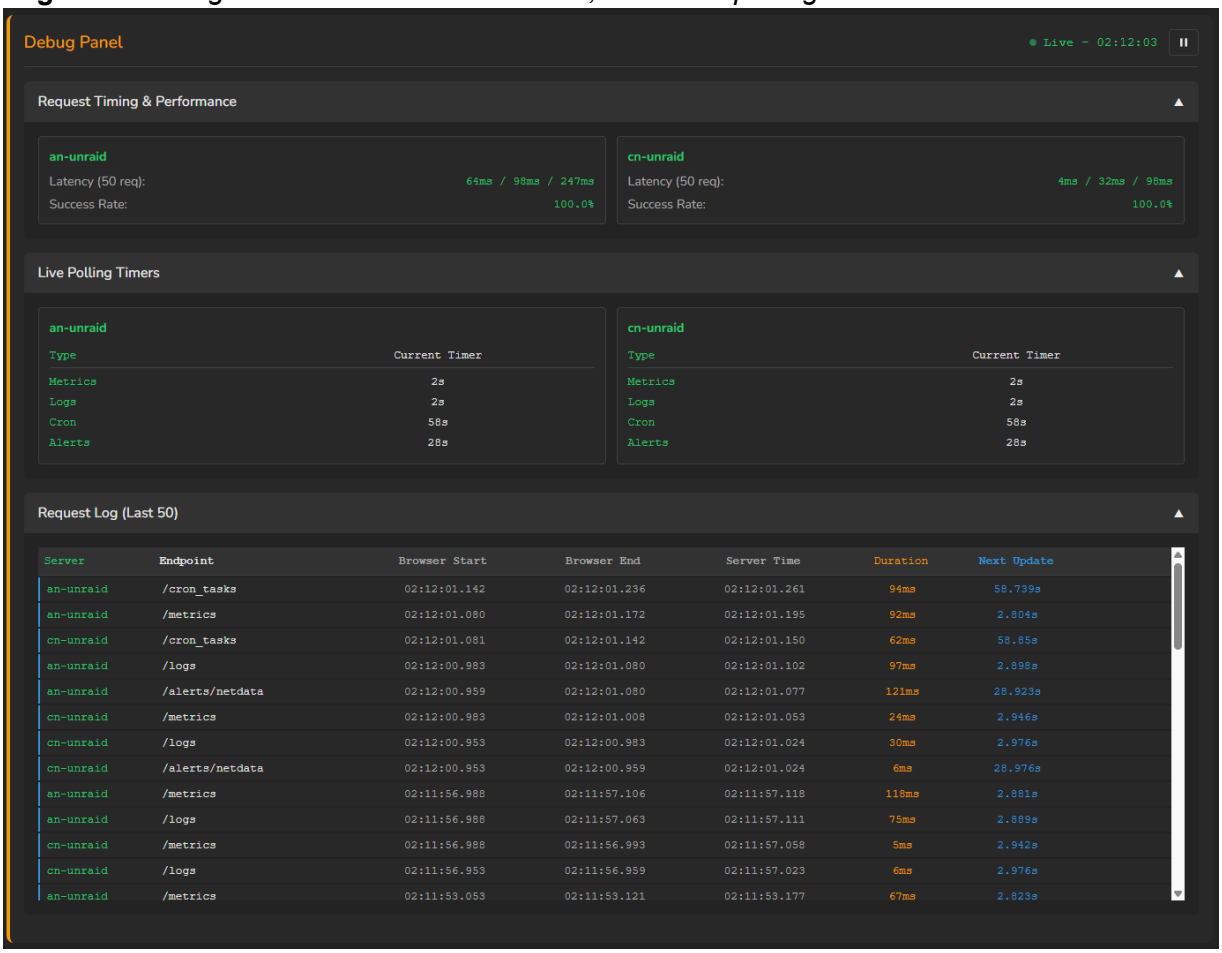
Platform-agnostic deployment while leveraging cloud services required careful architecture. Environment variables handle all deployment-specific configuration of the Web container. This enables identical containers to run locally or on any combination of platforms. Without a bundled VPN (Virtual Private Network) solution, port forwarding is used to expose API instances to the public Internet. For security, HTTPS-only communication and a valid API key are required by default to establish a connection between containers.

Lessons Learned and Future Directions

Key technical lessons include the importance of container optimization (using Python's "slim" containers proved useful), the value of simplicity (avoiding databases reduced complexity and enhanced portability), and that automation saves significant time while guarding against human error. Importantly, Google Cloud Platform does not allow EXEC access for container debugging, requiring alternative troubleshooting approaches and local development.

Architectural decisions that were validated include microservices enabling independent scaling, and early debug tool investment accelerating development. The hinting system demonstrates how simple solutions can eliminate entire cascades of bugs (timing-related ones, in this case).

***Image 12** - Debug Panel within DashPad-Web, shows all polling information and HTTPS traffic:*



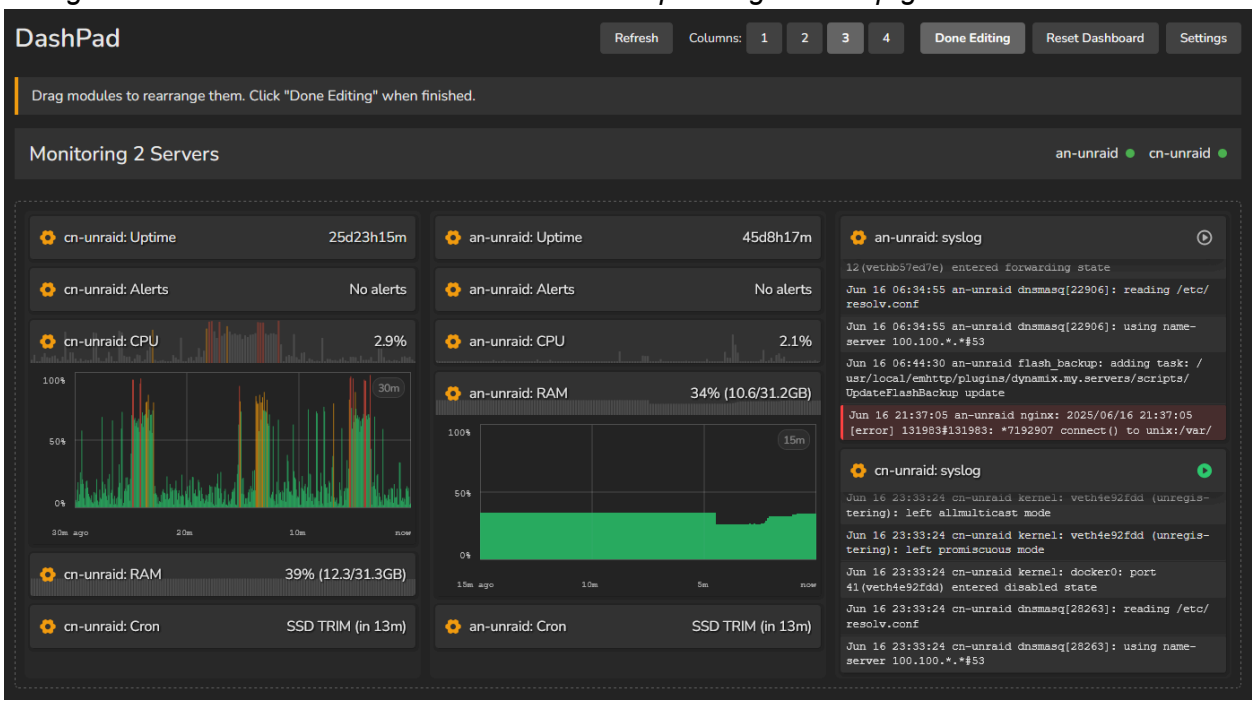
Future enhancements will include a notification system, providing browser-based alerts and optional sound alerts with configurable triggers. Docker container monitoring and additional module types are also planned. The modular architecture supports many of these additions without core modifications.

Documentation and source code for both DashPad-Web and DashPad-API will become available pending further improvements and finalization. GitHub Pages will host all materials at no cost, ensuring free online access for all users.

Conclusion

DashPad successfully demonstrates that cloud applications can be built with minimal resources and budgets with careful architectural considerations. The project breathes new life into unused tablets, turning them into valuable monitoring tools while providing practical system monitoring capabilities. The platform-agnostic design, optimized performance (0.1 vCPU, 15% of 128M memory), and customized, modular architecture provide a foundation for continued development.

***Image 13** - DashPad-Web interface with "Edit Layout" mode activated, which allows modules to be organized within and between columns with simple drag-and-drop gestures:*



Additional reading will be available at dashpad.neal.media and on GitHub.