Master Consolidation Plan for the Iraqi Election Platform

Overview & Context

You have accumulated multiple overlapping codebases (both local folders and GitHub repositories) for the "Hamlet" Iraqi Election Platform. This has led to confusion about which version is most up-to-date or suitable. The goal of this plan is to identify the best components (frontend and backend), compare each repository/folder’s status, and define a clear roadmap to consolidate everything into a single, production-ready platform. The emphasis is on preserving the modern social-media-style frontend (with features like stories, reels, posts) and the robust backend with the full candidate database.

Below we break down each relevant repository/folder, assess its purpose and completeness, and then provide a comparative readiness score and merge strategy. Finally, a step-by-step consolidation and deployment plan is outlined.

Repository & Folder Analysis

1. “Copy-of-Hamlet-social” (GitHub vs Local)

GitHub Repo (Copy-of-Hamlet-social) – This is a Next.js 14 application (TypeScript, Tailwind CSS) for the Iraqi Election Platform

GitHub

. It’s a full-featured frontend with bilingual support (Arabic/English), dynamic routing for candidates, dark/light mode, etc.

GitHub

GitHub

. The code includes React components for a social feed (stories, posts, reels) as well as election-specific pages (candidate lists and profiles). For example, it has components like Stories.tsx for Instagram-style story reels and PostCard.tsx for social posts. This frontend is highly complete – key features implemented include: multi-language support, RTL layout for Arabic, social media feeds (Posts/Reels/Events), candidate profiles, and even a “Serious” tab for civic data. In short, it matches the desired “purple theme, glass-morphism design” with social features you described. The current state uses some mock data (in a constants.ts) but is “90%” ready, needing only to connect to the real API endpoints.

Local Folder (E:\HamletUnified\Copy-of-Hamlet-social) – This appears to be an older prototype of the frontend with an attractive UI design. It contains ~6,484 files (~160 MB) including an index.html, CSS, JS, etc., and uses Tailwind via CDN. It features a beautiful glassmorphism UI (teal/cyan accents by default, easily changed to purple) and social media-style elements. Essentially, this was a standalone static frontend demonstrating the desired design aesthetic for the social app. However, it is not a modern React app and is built with a simpler tech stack (likely static or minimal framework). All the visual design elements from this prototype can be carried into the Next.js app (since the Next.js app also uses Tailwind and has a theming system). In fact, the Next.js repo already embraces that design: it has a color theme system (e.g. a default “euphratesTeal” theme with the teal color, which can be switched to a purple theme easily in code) – so the essence of this prototype has been merged into the React app. Conclusion: The GitHub Next.js codebase is the one to focus on (it’s more up-to-date and maintainable), while the local static folder is more of a UI reference. The GitHub repo “Copy-of-Hamlet-social” should be considered the canonical frontend source going forward, offering the complete social + election features in one. (Note: There is another repository listed as “Copy-of-Hamlet-social-” with a trailing dash. It appears to be either a duplicate or an empty placeholder – no significant code was found there, so we can ignore it.)

2. “hamlet-unified-complete-2027”

GitHub Repo (hamlet-unified-complete-2027) – This repository represents an attempt to unify the “social” and “civic” frontends into one. The README indicates it hosts the social experience and also embeds the full civic election interface under a “Serious” tab

GitHub

. The project is a React/TypeScript SPA (not using Next.js but rather an import-map approach) where the social shell wraps the election dashboard components

GitHub

GitHub

. For example, it mentions a SeriousExperience.tsx that loads the civic app, and shared API client/types between the two modes

GitHub

. This repo thus documents how the integration can work: the file structure shows components/views for candidates, profile, compose, etc., similar to the social app, and a components/serious/ for the civic modules

GitHub

GitHub

. However, this GitHub version is likely a snapshot or reference – according to the analysis, your local code in E:\HamletUnified\ (the Next.js app) is actually more up-to-date than this repo. In fact, your local “hamlet-unified-complete-2027” folder turned out to be empty (just an .env and import script), which caused confusion. Essentially, the code that the name suggests is present in this GitHub repo, but locally that folder was just a placeholder. Conclusion: Treat this repo as a reference documentation of how the social+civic integration is structured, but continue development on the primary Next.js frontend (which already includes those integrated features). The local root Next.js app corresponds to this but is newer. We will use the local/GitHub “unified” frontend concept (social shell + serious tab) in our final product, but we don’t need to separately merge anything because the main frontend already contains these components.

3. “hamlet-platform-nextjs” vs “New-Repository-hamlet-platform-nextjs”

hamlet-platform-nextjs (GitHub) – This repository is basically the same Next.js 14 frontend described above, under a different name. The README and structure are virtually identical to “Copy-of-Hamlet-social” – Next.js 14, bilingual, with candidates pages, etc.

GitHub

GitHub

. It was likely an earlier or alternate upload of the frontend code. (For instance, it even includes instructions to start the backend on port 4001 and run smoke tests

GitHub

GitHub

, indicating it pairs with the same backend). Conclusion: The code in this repo duplicates what’s in Copy-of-Hamlet-social. It’s not necessary to maintain both. We should consolidate to one canonical frontend repo. Since “Copy-of-Hamlet-social” contains additional documentation (the comprehensive artifact/guide) and presumably the latest tweaks, it can be our primary. “hamlet-platform-nextjs” can be archived or left aside to avoid confusion, as it doesn’t contain anything unique.

New-Repository-hamlet-platform-nextjs (GitHub) – This is an empty or minimal repo (it only has a README stub

GitHub

). It seems no actual code was added here. We can ignore this repository entirely. It has no impact on the consolidation because it doesn’t provide any code or updates.

4. “-hamlet-production-d” (GitHub)

This repository (the name suggests “Hamlet Production”) is extremely important – it contains the fully integrated project with both backend and frontend subdirectories, intended for production deployment.

Backend (-hamlet-production-d/backend) – This is the “Primary Backend” for the platform. It’s a Node/Express application with a PostgreSQL database via Prisma ORM. The Prisma schema defines a comprehensive Candidate model (over 25 fields covering IDs, names, party, governorate, contact info, verification status, stats like views/supporters, timestamps, etc.)

GitHub

. This matches the real data of thousands of candidates. The backend code is robust: it includes security middleware (helmet, cors, rate limiting) and logging (morgan, winston)

GitHub

GitHub

. Key API endpoints are implemented for candidates and related data. For example, /api/candidates returns paginated candidates from the database (using Prisma queries)

GitHub

GitHub

, and other endpoints like /api/search, /api/governorates, /api/parties, /api/stats, /api/trending are defined to retrieve aggregated stats

GitHub

GitHub

. This backend expects a real database connection (Supabase PostgreSQL, as configured via .env). It is nearly production-ready – Claude’s audit rated it 85% ready, with the main missing piece being actually connecting it to the Supabase DB and importing the data. All necessary scripts for importing candidates are present (e.g., import-candidates.js, data cleaning scripts). In fact, 7,769 real candidate records have been consolidated for import

GitHub

GitHub

, and the schema is ready to accommodate them. The backend also has placeholders for social media features: there’s a social.ts route file that defines endpoints like /social/posts, /social/events, etc., which currently use in-memory mock data structures

GitHub

GitHub

. This is great for now – it means the structure for posts, reels, events, follow/like actions is in place

GitHub

GitHub

, but those will need to be backed by a database or at least some persistent store later. Conclusion (Backend): This is the backend to use moving forward. It’s modern, secure, and aligned with the data. We will focus on this codebase (and the corresponding E:\HamletUnified\backend folder which is the same content) as the sole backend, and retire older backend attempts.

Frontend (-hamlet-production-d/frontend) – In this production repo, there is also a frontend folder. According to the README, it’s a Next.js 14 app (TypeScript, Tailwind, etc. just like our primary frontend)

GitHub

. This likely mirrors the code from the other frontend repos (perhaps a copy of the same Next.js app). In other words, “-hamlet-production-d” is structured as a monorepo containing both the Next.js frontend and the Express backend configured to work together. The README’s quick start confirms this: it instructs how to start the backend on port 4001 and the frontend on 3000

GitHub

GitHub

. There may not be significant differences in the frontend code here versus the “Copy-of-Hamlet-social” repo, aside from possibly configuration tweaks for production. Given that we want a unified codebase, we can treat this “production” repository as the integration point. In practice, you have the same on your local machine: E:\HamletUnified\ (frontend) and E:\HamletUnified\backend\ (backend) which correspond to these. Conclusion (Frontend): Use the Next.js app (from this repo or the Copy-of-Hamlet-social repo – they should be synchronized). We will ensure that whichever copy is latest, we merge changes so that the production repo contains the latest frontend code as well. The result will be one combined project (frontend + backend), either kept as a single repository or two coordinated repos. For simplicity, consider consolidating into the “hamlet-production” project structure, since it’s already set up for both.

5. “hamlet-complete-mvp”

This was an earlier attempt at a full-stack MVP. It has a backend (Express) and possibly some frontend code or documentation, but it is largely obsolete now in light of the above. Notably, the hamlet-complete-mvp/backend uses hardcoded mock data – it generates 200 dummy candidates in memory

GitHub

and serves them via very basic endpoints

GitHub

GitHub

. It has no real database connection or Prisma, and only trivial logic (for example, filtering and pagination of the in-memory array)

GitHub

GitHub

. This aligns with Claude’s finding that it was a simple placeholder used for initial testing. There is a documentation folder in this repo with various notes, but any useful info from that has likely been superseded by the more thorough artifact in the newer repos. Conclusion: We will avoid using hamlet-complete-mvp code. Its backend is a less-developed version of the production backend, and it lacks all the real data integration. Its presence has been a source of confusion, so it’s better to archive it. The only value here was perhaps some deployment configuration or CORS settings which we have already incorporated into the production backend (indeed, the production backend’s CORS config includes references to the old Vercel URLs from this MVP

GitHub

, so nothing is lost).

6. “iraqi-election-platform”

This repository is a standalone front-end dashboard – essentially a static website (HTML/CSS/JS) for monitoring election statistics

GitHub

. It features real-time updates of stats, charts of turnout by region, etc., all implemented in pure JavaScript and HTML5

GitHub

GitHub

. Importantly, it does not share any code with the main Hamlet platform – it doesn’t use the database or backend, and was meant as a lightweight dashboard. While it’s a neat implementation (e.g., it updates every 10 seconds and has an interactive UI

GitHub

), it is out of scope for the unified platform MVP. Our main platform already has an analytics section (for example, the Next.js app can display stats like total candidates, by governorate, etc., via its /api/stats endpoint

GitHub

GitHub

). If needed, we can later incorporate some ideas from this dashboard (such as dynamic updates or visualizations) into the main app. But for now, we will not merge this with the main code – it stands as a separate project. We might keep it archived or running separately for real-time election night monitoring, but it’s not required for the candidate social platform. Conclusion: No action needed on this repo for consolidation; it’s separate. Focus resources on the main platform’s front/back integration.

7. Other Local Folders and Assets

In addition to the above, you have several local directories that were part of earlier consolidation efforts. We briefly note them and their relevance:

E:\HamletUnified\ (Root) – This is the primary frontend codebase (Next.js app) that we’ve been discussing. It contains all the React components (139+ components, including many views for social and civic features). This will be our main frontend going forward. (It’s essentially the same as Copy-of-Hamlet-social/hamlet-platform-nextjs code on GitHub, with the latest updates.) It’s already been deployed to Vercel as an MVP (using mock data). We will connect it to the live backend next.

E:\HamletUnified\backend\ – The primary backend codebase (Express/Prisma). This corresponds to the “-hamlet-production-d/backend” on GitHub and has everything described earlier (schema, scripts, etc.). This is our main backend moving forward.

E:\HamletUnified\shared-schema\ – This likely contains shared TypeScript types/interfaces intended to be used by both front and backend (e.g., User, Post, Candidate types). In the integration repo, the code imports types from a shared schema (we saw references like import { Post, User } from 'shared-schema/types' in the backend)

GitHub

. We must ensure these types are available to both sides – possibly by merging this folder into the project (or copying the type definitions into the frontend and backend codebases). Maintaining a single source of truth for data models is ideal. Conclusion: Integrate the shared-schema types into the final project to avoid type mismatches. This will help front and back agree on data structures (for example, ensuring the frontend’s expectations of a Candidate or Post object match the backend’s).

Consolidation Folders (full\_consolidation\frontend, frontend-aigoodstudio, missinggold\_fresh\_structure\new-frontend, etc.) – These appear to be remnants of attempts to restructure or experiment with the frontend. For instance, “frontend-aigoodstudio” might contain some design output from a studio or an AI tool, and “missinggold-fresh” sounds like a codebase possibly for an event or campaign named “Missing Gold” (maybe unrelated to the election platform). Given that our chosen frontend is already feature-rich and well-designed, we do not need to incorporate these miscellaneous fronts. At most, if there are any unique UI components or ideas in them, we could manually cherry-pick those. But there is no indication of something critical there. Conclusion: No merging of these is required – they are likely outdated or specialized. We will focus on the main Next.js front and ignore these extras to avoid distraction.

E:\HamletUnified\asset-completeredrive\HamletUnified\ – This is essentially a backup copy of your entire project (1.4 GB of duplicate files). It contains nothing new – just an older snapshot. It has caused confusion due to duplicate content. Conclusion: Leave this as an offline backup; do not use it for development. We’ve solved the “treasure asset” mystery: it doesn’t contain any unique “goldmine” of assets, it’s just duplicative. So we can safely ignore the asset-completeredrive contents during consolidation (to avoid accidentally working on outdated copies). All real development should happen in the main directory (which we have identified clearly now).

By analyzing all these sources, we conclude that the most developed and up-to-date components are: the Next.js frontend (social-enabled, purple theme) and the Express/Prisma backend (with real DB schema). Most other versions are either duplicates, stubs, or older iterations. Therefore, our consolidation will center around merging those two and making sure nothing of value from the others is lost.

Below is a quick comparative summary of each component’s readiness and our decision:

Next.js 14 Frontend (“Hamlet” social platform) – Readiness: ~90% (production-quality UI, just using mock data currently). Features: Full social media interface (stories, posts, profiles, etc.) plus election browsing. Action: Keep as primary frontend; integrate with backend API and ensure design/theme meets requirements.

Express/Prisma Backend (Supabase DB) – Readiness: ~85% (complete schema and API routes for candidates; needs DB connection and to flesh out social endpoints)

GitHub

. Features: Real data model for ~7,700 candidates, secure API, prepared scripts. Action: Keep as primary backend; connect to database and import data; extend endpoints for social features as needed.

“Copy-of-Hamlet-social” static prototype – Readiness: 100% for UI design (it’s a fully built static site) but 0% integrated (no dynamic data). Features: Beautiful design, but codebase not reusable directly (not React). Action: Archive the code; borrow any needed CSS/HTML ideas. (The Next.js app already incorporates its look and feel, so nothing major to do here).

“hamlet-complete-mvp” backend – Readiness: ~20% (basic Express with mock data only)

GitHub

. Features: Very minimal; lacks DB, not production. Action: Discard for consolidation (all functionality replaced by the new backend).

Unified Frontend (hamlet-unified-complete-2027) – Readiness: conceptually 80% (code present in GitHub), but local folder was empty. Features: Combined social+civic UI, but likely older snapshot. Action: Superseded by the current Next.js front (which already contains both aspects). Use as reference if needed.

Misc. frontends (test-new-frontend, missinggold, etc.) – Readiness: unknown/low. Action: No merge needed. Possibly skip or archive.

Standalone Dashboard (iraqi-election-platform) – Readiness: 100% as a static page. Features: Real-time stats (simulated). Action: Keep separate. We may integrate some stats into the main app differently; no direct code merge.

(The table below, from the audit, highlights the key assets and their readiness: e.g., the frontend is ~90% ready and just needs API hookup, backend ~85% ready needing DB setup).

Consolidation & Deployment Roadmap

With the core pieces identified, here are the clear steps to consolidate everything and move towards deployment:

Step 1: Set the Canonical Codebase

Establish one source of truth for each tier of the application:

Frontend: Use the Next.js 14 application located in E:\HamletUnified\ (or the latest GitHub repo containing it, e.g. “Copy-of-Hamlet-social”). Ensure that any small differences between the GitHub repo and local version are merged so that we have the latest code in one place. This includes merging in the language translation files (Arabic, Kurdish, English JSON dictionaries)

GitHub

and any UI tweaks. From now on, all frontend development will continue in this unified codebase.

Backend: Use the Express/Prisma backend in E:\HamletUnified\backend\ (as reflected by “-hamlet-production-d/backend”). This is our official backend. Remove/ignore other backend folders to avoid confusion (no more switching between multiple backend directories). The backend code should reside in one folder/repo moving forward. If not already done, add the shared-schema\types.ts definitions into this backend project (for example, as a Git submodule or just copying the file) so that the import { ... } from 'shared-schema/types' references resolve properly

GitHub

. Alternatively, we can adjust those imports to point to a local types.ts file that both frontend and backend share.

At this point, our working directories should be essentially:

/frontend (Next.js app with social+serious features)

/backend (Node/Express app with Prisma, connected to Supabase)

/shared (shared types, if we keep it as separate folder or merge into front/back)

This matches the structure implied in the production repo and will simplify deployment (frontend and backend can be deployed separately but originate from a cohesive project structure).

Step 2: Connect Frontend to Backend API

Right now, the Next.js frontend is using mock data (stored in a constants file) and not calling the real API yet. We need to wire it up to the live backend endpoints:

Configure API URL: In the frontend code, set the environment variable NEXT\_PUBLIC\_API\_BASE\_URL to the base URL of the backend. In development, this would be http://localhost:4001 (as noted in the README)

GitHub

. In production, this will point to the deployed API (we will update it when deploying). Ensure this is placed in .env.local and used by the frontend’s API service module (the repo has a file like lib/api.ts or similar for Axios)

GitHub

.

Replace Mock Data Usage: Identify where the frontend currently imports or uses constants.ts or any fake data. According to the architecture, the main feed and profile components rely on these mocks for now

GitHub

. We will replace those with real API calls. The lib/api.ts (Axios client) should be used to fetch data from endpoints: e.g., GET /api/candidates for candidate lists, GET /api/stats for statistics, GET /api/trending for trending candidates, etc., which our backend provides

GitHub

GitHub

. Also, for social features: use GET /social/posts, GET /social/events, etc., instead of static arrays. Initially, since the backend’s social routes still return mock data from memory

GitHub

GitHub

, this will function similarly to the frontend’s previous constants (just now the mock comes from server instead of client). This is fine for an MVP – users can create posts or events which will live in server memory during the session

GitHub

GitHub

. The key is the frontend will be exchanging data via API rather than using a local fake source.

Test Integration: Run the backend locally (npm start in /backend, which listens on port 4001)

GitHub

and the frontend (npm run dev in /frontend, port 3000). Open the app and verify that data loads from the backend:

The home page should load actual candidate data from the database via the API. Once the DB is populated, you should see real candidate names, etc., instead of placeholder data.

Filters (governorate, party, etc.) should trigger API queries (the backend supports query params for these filters on candidates)

GitHub

GitHub

.

The statistics section should use /api/stats – verify it shows correct totals (once data is in).

Try the social features: create a post (the frontend will call POST /social/posts which in the backend currently adds to the posts array and returns the new post

GitHub

GitHub

). Ensure this flows without errors. Similarly, test viewing reels, events, etc., to see that the endpoints (/social/events, etc.) return the mock lists from the server. This confirms that the front-back wiring is correct. We essentially achieve the same behavior as before, but now through a real API boundary. This is crucial for later deploying the frontend and backend separately.

Step 3: Database Setup and Data Import

With the integration in place, focus on getting the real data into the system:

Supabase/PostgreSQL: According to your .env and Claude’s report, you have a Supabase database ready (the connection URL and keys were identified in the audit). Confirm that this database is created and accessible. It’s currently empty (no tables or data). Use Prisma migration to create the schema in the DB: there should be a Prisma migration or you can run npx prisma db push using the schema.prisma to create the tables (Candidate model, etc.) on Supabase. Once done, verify the Candidate table exists.

Data Import: Use the provided import scripts to load the candidate data. The backend/import-candidates.js script is designed to take the CSV/JSON of candidates and insert them into the DB. First, ensure you have the final dataset ready. The audit mentions candidates\_production\_ready.csv (~41,000 records, which likely includes historical or multiple election cycles) and a smaller set of ~7,769 candidates (perhaps the current election’s candidates)

GitHub

GitHub

. Determine which dataset is needed for this MVP (7,769 seems to be the current election candidate count). Run the import script (with node or npm script if defined). This will populate the Candidate table. The script likely uses Prisma or direct queries to insert records. If it’s large, be patient (the plan estimated ~10-15 minutes for 41k records). After import, connect to Supabase or use a Prisma studio to verify that candidate entries are indeed in the database.

Verify Backend with Data: Restart the backend and test some endpoints:

GET /api/candidates?limit=5 should now return real candidate objects from the DB (with fields like fullNameArabic, party, etc.)

GitHub

GitHub

.

GET /api/governorates should aggregate governorate counts from real data

GitHub

.

The front-end, when refreshed, should display real information – e.g., the candidates page listing actual names from the database instead of “Ahmed Al-Maliki 1” etc. This is a major milestone: the platform is now backed by real data.

User Accounts & Social Data: In the current scope, user accounts (Voter/Candidate login) and social posts are still using stubbed data. If time permits, you might set up minimal persistence for these as well:

For users: The backend defines a /auth/login route in the API contract expecting a role and returns a user + token

GitHub

, but likely not implemented fully. You could create a simple User model in the database or just use a hardcoded user list for now. Since social features (like posting or following) require an authenticated user context, ensure the frontend’s login flow at least works with a dummy user (perhaps always logging in as a test user without password). This can be enhanced later.

For posts, events, etc.: To truly persist these, you’d define Prisma models for Post, Event, etc., and adjust the /social/\* routes to read/write to those tables. This might be beyond MVP scope, so as an interim, using the in-memory approach is fine – just understand that after server restart those entries reset. Given the focus is on candidates and viewing content, this is acceptable for now. Mark this for future improvement (perhaps a Phase 2 after MVP launch to add full social DB integration).

Step 4: Quality Assurance & Refinement

Before deployment, do a thorough run-through of the consolidated application:

UI/UX Pass: Ensure the frontend design still looks as intended with the new data. The theme should be purple (if not, adjust the Tailwind theme or default active theme to your preferred color – the audit noted it was easily changed from teal to purple). Verify bilingual support: switch the locale (e.g., the site likely auto-detects or has a language toggle). Arabic text should appear properly and layout should flip to RTL. Check that all major views load (Home, Candidates list, Candidate profile, any “Discover” or “Dashboard” sections, etc.). Check responsiveness on mobile sizing. Since this is a Next.js app, also ensure that SEO tags or dynamic meta (if any) are correct.

Functionality Test: Exercise all features:

Filter candidates by governorate/party (both via UI and via direct URL if applicable) – confirm the filtering works against real data.

View candidate profile pages – ensure all fields (name, bio, etc.) are populated if available in the data. Any field that’s null should be handled gracefully (no undefined text showing).

Use the social features: try posting a comment or creating a “reel” (if the UI allows uploading an image or just text) and ensure it appears in the feed. Even if this is not persistent long-term, it shows the end-to-end connectivity works (frontend -> API -> backend logic -> response -> frontend update). The social feed may also incorporate “trending” candidates or content – verify that if implemented.

Check the “Serious” tab (if present in the UI) which presumably shows a more traditional dashboard or stats. This likely uses the integrated civic components (from the hamlat-frontend-6-10). Ensure that view can render – it might be loading some mock data or using the same API in a different way. If any part of the serious dashboard expects certain endpoints (like those listed under /civic/\* in the API contract

GitHub

GitHub

), see if they are implemented or need stubbing. The API contract suggests endpoints for civic stats, etc. If those aren’t implemented yet, either implement simple versions or adjust the front to use the already available stats endpoints.

Multi-user flows: If a user logs in as a Candidate vs as a general user, does the UI change appropriately (e.g., a candidate user might see a different dashboard tab)? The code in App.tsx hints that if a logged-in user is a Candidate, it navigates to a Dashboard view

GitHub

. Test this by simulating a login with role Candidate and ensure the Dashboard (perhaps CandidateDashboardView.tsx) loads. This might require adjusting the dummy login to let you choose role.

Performance Considerations: Next.js 14 (App Router) should handle performance well. Just ensure that any heavy data isn’t fetched all at once unnecessarily. The backend endpoints are paginated for candidates

GitHub

GitHub

, so the front should be requesting in sensible chunks. Also check for any memory heavy operations. Tailwind and React usage seem fine from the audit. If you imported 41k candidates but only ~7.7k are needed, confirm if the data set can be limited to current election to save on initial load.

Security: The backend has security middleware (Helmet, CORS, rate limiter) in place

GitHub

GitHub

. That’s good. Ensure the CORS allows your deployment domains (currently it was allowing some Vercel URLs and localhost)

GitHub

. Update the allowedOrigins list in the backend config to include your production frontend domain once known. Also, since user auth isn’t fully implemented, consider disabling any admin or critical actions or documenting that the MVP has limited auth. No actual sensitive data is involved beyond what’s public.

Step 5: Deployment Plan

With a consolidated and tested codebase, proceed to deployment:

Backend Deployment: Choose a hosting for the Node/Express API. Since you use Supabase for the database, a convenient choice is to deploy the Node server on a service like Render, Railway, or even as a Vercel serverless function (though long-lived Express might be better on a dedicated service). The audit suggests Render as an option. Ensure the environment variables on the server (notably DATABASE\_URL for Prisma, and any JWT secret or Supabase URL if needed) are set. Build and run the server on the host – it should connect to Supabase and start listening on some URL (Render will give you a URL). Test the /health endpoint if one exists (the Claude recommendation text file had a health URL, possibly indicating a health check) to confirm it’s up. All API routes should now be accessible publicly. Double-check that HTTPS CORS is configured to allow your frontend origin. If using Render, you might add the Vercel domain to CORS.

Frontend Deployment: Deploy the Next.js app on Vercel (since it’s a Next.js 14 project, Vercel is ideal). You likely already have it connected (the audit mentioned it was deployed on Vercel but just needed the backend). In Vercel settings, set the NEXT\_PUBLIC\_API\_BASE\_URL to the backend’s deployed URL. Trigger a build and deploy. Once live, test the front-end URL – it should fetch data from the backend (you might need to enable CORS on the backend for the Vercel domain, as noted). The site should function the same as locally, showing real data.

Post-Deploy Testing: On the production site, walk through everything again. Particularly, verify that API calls from the front-end succeed (open dev tools console – no CORS errors or 500 errors). The “Ready!” message on the backend console indicates it connected to the DB successfully

GitHub

– ensure you see that in logs. Test creating a post or event on the production site – these should hit the live backend and get responses. (They’ll still be non-persistent beyond memory, but that’s expected until we add a database for them.)

Monitoring: The backend includes morgan logging and likely prints out when it starts. Use that to monitor usage. Supabase can show you if queries are coming in (and how many rows are in the Candidate table, etc.). Also, set up the health check if not already (maybe that / route with status is the health check

GitHub

; you can use an uptime monitor to hit it periodically).

Final Domain and Access: If you have a custom domain for the platform, map it to the Vercel app. The backend can remain on its provided domain (since only the front-end calls it). Ensure everything is secure (the backend doesn’t expose any sensitive endpoint without auth – currently, all data is public read-only, which is fine for public candidate info).

Step 6: Future Enhancements (Post-MVP)

While not needed immediately, keep in mind these next steps after initial launch:

Implement persistent social features: Add database models for Posts, Events, etc., and connect the /social/\* routes to those. This way, user-generated content is saved. You could use Supabase for storing these as well. Shared schema definitions for Post, Event exist (perhaps in the shared types) – utilize them.

User Authentication: Integrate a real auth system (Supabase Auth or custom JWT) so that users can register/login (especially candidates who want to claim their profile, or voters who want to follow candidates). Currently the app might allow switching user roles without true auth; adding this will make follow/like actions meaningful.

Content Moderation & Verification: Use the verificationStatus field in Candidate (it’s already in schema) to mark candidates as verified, etc., and expose that in the UI. Perhaps implement the “follow” and “like” endpoints fully to update counts in the DB.

Refine the Dashboard (“Serious” tab): If the embedded serious dashboard is still using mock data, connect it to real data (e.g., integrity reports, participation stats by governorate). This may involve implementing the /civic/\* API routes as per the contract

GitHub

GitHub

.

Deployment Scaling: Eventually, consider splitting the repo if needed (monorepo vs separate). For now, a monorepo is fine. Ensure both front and back are tracked in version control. Continue to keep them in sync when updating shared things (like types or API contract changes).

By following this roadmap, you will end up with a single, consolidated product that leverages the best of all your code sources. Specifically, you’ll have a modern frontend with all the social media features and a unified design, backed by a powerful backend with the full election candidate database. This addresses the original confusion (multiple versions) by definitively choosing the most complete implementations and focusing only on those. The priorities were clearly to get the social UI and real data working together – which we achieve by pairing the Next.js app (purple social interface) with the Prisma backend (7k+ candidates)

GitHub

.

All other repositories or folders either feed into these primary ones as references or are set aside to reduce noise. This consolidated platform is roughly 70-80% complete as noted, and our plan covers the remaining 20-30% to reach 100%. By executing the steps above, Claude (or any developer) can now proceed without confusion, following a canonical set of instructions and code.

Finally, here’s a succinct summary of what to keep vs discard for clarity:

Keep & Integrate: E:\HamletUnified\ frontend (Next.js social+serious), E:\HamletUnified\backend\ (Express/Prisma), and Supabase DB. These correspond to Copy-of-Hamlet-social/hamlet-platform-nextjs (front) and -hamlet-production-d (back) on GitHub – the latest and greatest code

GitHub

GitHub

.

Discard/Archive: Static prototype frontend (already mined for design), hamlet-complete-mvp (old backend with mocks), empty folders like local hamlet-unified-complete-2027 and hamlet-platform-nextjs (no code), and the standalone dashboard (use separately if needed). These will not be part of the main build or deployment.

Reference Only: GitHub hamlet-unified-complete-2027 (for understanding integration structure)

GitHub

and any docs (the artifact guide

GitHub

GitHub

, Claude’s audit) as documentation. They’ve informed our plan but the implementation work happens in the kept codebases.

With everything consolidated, you can be confident that all features are included and no important pieces are left behind. This plan gives a clear path to launch a unified platform that meets your requirements: a social-media inspired election hub, deployed and ready for user engagement. ✅

Sources:

Detailed inventory of local project folders and their contents.

Claude’s audit confirming which frontend/backend are production-ready and their feature completeness.

Excerpts from repository READMEs and code showing the tech stack and implemented features (Next.js app structure, Express API endpoints, Prisma schema)

GitHub

GitHub

GitHub

.

The “Hamlet Production” repository indicating the integrated approach with Next.js 14 frontend and Express backend for 7,751 candidates

GitHub

GitHub

.

API route definitions illustrating how the backend serves candidate data and prepared endpoints for social features

GitHub

GitHub

.

Static dashboard description for context on the out-of-scope separate project

GitHub

.

Sources