ChatGPT

Plan for an Internal Browser-Based Development Platform

Architectural Overview

The platform will be a **browser-accessible IDE** that mirrors Replit's functionality while integrating with the bank's infrastructure. Key components include a **web-based IDE frontend**, a **backend orchestration service**, isolated **development containers**, and an **AI assistant**. The **IDE frontend** (built with React) provides a Monaco-based code editor, file explorer, and a chat panel for the AI assistant, all running in the user's browser. The **backend service** (e.g. written in Node or Go) handles authentication, user sessions, project metadata, and orchestrates container lifecycles via an internal Kubernetes cluster. Each developer session launches a dedicated **containerized environment** (with Python, Node.js/Next.js toolchains pre-installed) on the Kubernetes cluster, providing isolation and reproducibility. User code is stored on persistent network-attached storage (or an object store like S3) and mounted into the container, so work isn't lost when containers stop [1] [2]. The web IDE communicates with the container via WebSocket or gRPC channels for real-time editing, terminal access, and previewing web applications [3]. This setup allows running Python scripts or Next.js development servers inside the container, with output streamed back to the IDE. A Kubernetes **Ingress controller** routes preview requests (e.g. opening a Next.js app in a browser tab) to the correct user container based on context [4]. Unused containers will be stopped after a period of inactivity to free resources (with a keep-alive for active work) [5]. Role-based access is enforced at the platform level: for example, certain user roles can only create sandbox projects, whereas others may deploy to staging or prod. The diagram below illustrates the high-level architecture and integration points of the platform:

High-level architecture of the internal browser-based development platform, showing the web IDE, backend services, containerized dev environments, AI assistant, and integration with auth and internal systems.

Recommended Tooling and Frameworks

Browser IDE & Editor: Use a web IDE framework like *Eclipse Theia* or *VS Code code-server* for a full-featured, extensible IDE in the browser. These support the **Monaco Editor** (the same editor engine as VS Code) for syntax highlighting, IntelliSense, and language server protocol (LSP) integration [6]. Theia, for example, can run as a web application split into a frontend (in-browser) and backend (Node.js), and can leverage VS Code extensions for Python and ESLint, giving developers a familiar experience. The editor should support both Python and JavaScript/TypeScript (for React/Next.js) with language servers (e.g. Pyright for Python, TypeScript LS for Next.js) to provide code completion and linting.

Container Orchestration: Use *Kubernetes* to manage user dev environments at scale [1]. Each project or session runs in its own container (based on a Docker image that includes Python, Node.js, and common build tools). Kubernetes provides isolation and can schedule containers on available nodes, auto-scale, and restart as needed. A lightweight alternative is using VM-based sandboxes, but containers are faster and more resource-efficient. Leverage Kubernetes features like Namespaces and NetworkPolicies to isolate network access per environment (e.g. only allow required internal API calls). Use a *Persistent Volume* or object storage (e.g. *AWS S3 or on-prem MinIO*) for storing code and assets, mounted into containers [1]. This ensures file persistence across restarts and facilitates backups.

Authentication & Access Control: Integrate with the organisation's existing **SSO**. For example, deploy an *OpenID Connect (OAuth2)* identity provider (such as **Keycloak**) that can federate with the corporate Active Directory/LDAP [7]. This way, users log in with corporate credentials via OAuth, and the platform receives JWT tokens asserting identity and group membership. The platform's backend will validate tokens and also query LDAP/AD as needed for group info (e.g. to determine roles like "developer", "approver", "admin"). All UI and API calls use this SSO – no separate passwords to manage. Authorisation is fine-grained: define roles and permissions for actions (e.g. who can create projects, who can deploy to production). Use built-in roles from the IdP or map LDAP groups to platform roles (for instance, only users in the "DevOps" group can approve deployments).

Development Frameworks: For supporting **Python**, include common libraries (pip, virtualenv or Poetry for package management, Jupyter support if needed) in the base container. For **React/Next.js**, include Node.js LTS, npm/Yarn, and frameworks like Next.js and create-react-app. Provide templates to quickly scaffold new projects (e.g. a template Next.js frontend or a Python Flask API). Use container images that mirror the organisation's approved tech stack – for example, an image with Python 3.x and Node 18, pre-configured with internal package registries. This ensures developers use sanctioned library versions and can access internal PyPI/NPM mirrors for third-party packages (to satisfy security vetting of dependencies).

Collaboration & VCS: (Optional) Integrate source control for persistence and collaboration. The IDE can integrate with the organisation's Git service (e.g. GitLab, GitHub Enterprise, or Bitbucket). Developers could sync their code to Git repositories directly from the browser IDE. Additionally, enabling collaborative editing (like Replit's multiplayer or VS Code Live Share) could be considered – for instance, Theia supports collaborative extensions, which would let multiple devs work in the same workspace, although this can be phased in later. All code changes should eventually reside in the central Git repos for audit and CI/CD integration.

Other Tooling: Include a built-in **terminal** in the IDE so developers can run commands (e.g. package installs, tests) inside their container. Utilize web terminal libraries (such as xterm.js) to render the container's shell in the browser. Also, integrate CI pipeline triggers – e.g. a button to run unit tests or static analysis – which can either run in the container or trigger external CI jobs. For Next.js specifically, a preview server can run on a random port; the platform's proxy (Ingress) will map a unique subdomain or URL path to that port so the developer can open the live preview in the browser securely.

Security, Compliance, and Governance

Building this platform in a financial services context requires strict **security and compliance** measures at every layer. The solution will be deployed inside the bank's firewall (on-premises or VPC), with no public access. Key measures include:

- **Network Isolation & Dependency Management:** Development containers should be on an isolated network. By default, deny outbound internet access from containers to prevent data exfiltration or unapproved downloads. Instead, provide controlled access to required resources: e.g. allow reaching internal APIs/services, and allow package installs only via approved internal artifact repositories (mirrored PyPI/NPM with vetted libraries). This ensures only approved open-source packages (scanned for vulnerabilities and license compliance) are used. If internet access is ever needed (e.g. pulling a new library), require an explicit exception or route through a secure proxy that logs and virus-scans downloads.

- **Container Security:** Use hardened base images (minimal OS, updated security patches). Run containers with non-root users and drop unnecessary Linux capabilities. Apply Kubernetes NetworkPolicies to restrict container communication (each dev container only talks to necessary services, e.g. source repo, package registry, and its own preview port). Enable resource quotas to prevent abuse (e.g. limit CPU, memory per container to avoid one user exhausting cluster resources). All data at rest (in persistent volumes or S3) should be encrypted. Data in transit (Web IDE to backend, backend to container) should use TLS – e.g. WebSockets secured via WSS, HTTPS for all API calls.

- **Authentication & Authorisation:** Enforce SSO login (no anonymous access). Use short-lived OAuth tokens and refresh them to avoid stale credentials. Leverage multi-factor authentication if the corporate auth supports it, especially for any production deployment actions. Authorisation within the platform should adhere to **least privilege** – e.g. a regular developer can use dev/test environments but only approved release managers can initiate prod deployments. Use groups/roles from the corporate directory to manage this, as noted above. For fine-grained project access, implement an internal permission model (e.g. project owners, collaborators, viewers) to control who can edit or view a given project's code.

- **Audit Logging:** All significant actions must be logged for audit trails – log user logins, container start/stop, code changes (could tie into version control commits), and deployment requests/approvals. Include what was deployed, by whom, and when. These logs should feed into the organisation's SIEM for monitoring. Also log AI assistant queries if they involve accessing internal knowledge or code, to monitor for any potential misuse (since the assistant could potentially be asked to reveal sensitive info – it must be prevented, see below).

- **Compliance & Code Governance:** Incorporate compliance checks into the development workflow. For example, integrate **static code analysis** and security scanners that run either continuously or on demand. The platform can automatically run linting and security checks on code save or commit – flagging issues like hardcoded secrets, use of banned functions, or OWASP Top 10 vulnerabilities. Tools such as SonarQube or Codacy (self-hosted) could be integrated to provide immediate feedback within the IDE. This can be supplemented by the AI assistant as well, which can highlight potential security issues in real-time (similar to Ghostwriter's proactive bug detection [8]). Enforce that any code to be deployed passes all tests and static analysis checks.

- **Approval Workflow for Deployments:** In a financial organisation, releases often require formal change control. The platform will include a gated deployment pipeline. When a developer is ready to deploy an app (frontend or service), they will **submit a deployment request**. This triggers an automated build/test (ensuring everything is green) and then awaits approval. The platform can integrate with existing change management systems or use an internal approval module – e.g. designated approvers get notified (via email or within the platform) and can approve or reject the deployment. Only upon approval will the platform actually release the code to the target environment (staging/prod). This process ensures compliance with regulatory requirements for code review and separation of duties.

- **Data Protection:** If any sensitive data is used in dev/test, ensure it's anonymised or masked according to internal policies. The platform itself should not store production secrets in dev containers. Provide a mechanism for managing secrets (like API keys for internal services) – e.g. integration with a secrets vault. Developers can request a token for an internal API which is scoped to their identity and project, and the platform injects it as an environment variable into

the container (and rotates it regularly). This way, dev containers can call internal APIs securely without hardcoding credentials.

- **MCP Server Governance:** The mention of **MCP servers** (Model Context Protocol) implies the organisation may expose certain data/tools via MCP for AI usage. We will ensure that any connections to MCP servers (for either the AI assistant or developer's code) are done securely and in line with governance. MCP servers themselves typically enforce their own security (they are lightweight APIs exposing data with context for LLMs [9]). We will maintain a registry of approved MCP servers and require that any AI agent action through MCP is auditable and authorized. For example, if an MCP server allows querying customer data, only certain roles in the platform (and perhaps only the AI assistant in a read-only documentation mode) can call it, and every call is logged. This prevents misuse of powerful AI-agent actions via MCP.

In summary, security is baked into every layer: from container sandboxing and network controls, to SSO auth, to enforcement of code quality and approval gates. This ensures development via the platform adheres to the bank's compliance requirements while still giving developers flexibility to build and test applications quickly.

AI Assistant (Embedded Chat Interface)

A standout feature is the **AI coding assistant**, embedded as a chat interface in the IDE (comparable to Replit's Ghostwriter). This assistant will help developers with code generation, auto-completion, documentation lookup, and validating code against internal standards. Crucially, the AI will be **grounded in the organisation's internal knowledge base** – it will know about the bank's APIs, data schemas, and coding guidelines so that its suggestions are contextually relevant and compliant.

Model and Hosting: To maintain data privacy, the AI model will be hosted internally (no calls to public APIs with proprietary code). We can leverage a large language model tuned for coding, such as an open-source code model (e.g. Code Llama or StarCoder). For high quality, a fine-tuned model (possibly based on Code Llama 34B or a smaller model like Mistral 7B) can be used. Fine-tuning the model on the organisation's code and documentation will align it with our coding style and APIs [10]. Another approach is using a managed service like Azure OpenAI (with GPT-4) if allowed, but an on-prem model avoids sending data outside. The chosen model will be integrated into the platform via an AI microservice – the IDE chat panel sends user prompts to this service and returns the assistant's answer or code suggestion.

Retrieval-Augmented Generation (RAG): To ensure the assistant's answers are up-to-date and accurate with internal knowledge, we will implement a RAG pipeline [11]. All relevant internal documentation (API specs, system design docs, coding standards, etc.) will be indexed into a vector database. When the user asks a question or when the assistant needs to autocomplete code involving an internal API, the assistant first retrieves relevant snippets from these documents. Those snippets are then provided to the LLM as context for generation. This grounding prevents hallucination and bases the answer on real data [11]. For example, if a developer asks "How do I call the internal payments API to initiate a transfer?", the assistant will retrieve the relevant API documentation page from the index and use it to provide the correct code example and explanation. This way, the assistant acts as a smart documentation search in addition to code generator.

Capabilities: The AI assistant will offer several key capabilities:

- **Code Completion & Generation:** As you code, it can suggest the next line or block (like Copilot/Ghostwriter). It uses the context of the current file and project to make relevant suggestions [12] [13]. Developers can also explicitly prompt it (via the chat) to write a function or component. For example: "*Write a Python function to calculate VaR (Value at Risk) given these parameters…*". The assistant will generate code that follows the bank's patterns (perhaps using approved libraries, and proper error handling as per internal standards).

- **Contextual Help & Documentation:** Developers can ask questions in natural language: "*What does error code X mean from the trading API?*" or "*Show me an example of using the customer data service*". The assistant will use RAG to fetch answers from internal docs and provide an explanation or code snippet. It essentially serves as a chatbot interface to the company's technical knowledge.

- **Enforcing Standards & Best Practices:** The assistant will be aware of internal coding standards (we can feed it the secure coding guidelines, style guides, etc.). When generating code, it will prefer compliant patterns. Moreover, developers can ask it to **review code**: e.g. "*Review my code for any security issues or deviations from best practices.*" The assistant can analyze the code (with the help of static analysis outputs if available) and point out issues like SQL injection risks or usage of disallowed functions, citing the specific internal rule it violates. Because the LLM has been fine-tuned or provided context with our standards, it can validate implementations against those benchmarks. This is akin to having an automated code reviewer that knows the company's rules.

- **AI Pair Programming:** The assistant works in real-time, as a pair programmer. As the developer types, it may offer a completion or highlight a potential bug. Replit's Ghostwriter, for instance, can proactively identify bugs and suggest fixes [8]. Our assistant will similarly leverage both the LLM's reasoning and perhaps integrate with linters to flag errors. For instance, if a developer is writing a React component and forgets to handle an error state, the assistant might suggest adding that based on internal UX guidelines.

Implementation Approach: We will likely adopt a **hybrid strategy**: use retrieval augmentation for factual accuracy and an LLM fine-tuned on code for fluent generation. The AI service can be built using frameworks like **LangChain** or **LlamaIndex** to manage the RAG workflow – i.e. embedding documents, performing similarity search on queries, constructing prompts, and caching responses. The vector database could be an open source solution like *Milvus* or *FAISS*, hosted internally, containing embeddings of all relevant text (code samples, API docs, etc.). Fine-tuning can be done via frameworks like HuggingFace's training pipelines or Low-Rank Adaptation (LoRA) on a base model with our data, which is a one-time (or periodic) effort to improve base knowledge of internal APIs.

For the chat UI, it will be integrated into the IDE frontend – possibly as a sidebar or panel. The interface will allow switching between chat Q&A mode and code completion mode. The chat can have presets like "Explain this code" (where it takes the selected code and explains it in plain English) or "Generate documentation" for a function, etc., to assist developers. All AI interactions will be **read-only with respect to production data** – i.e. the AI can fetch documentation or sample data schemas but it cannot execute actions that change data. This constraint, plus thorough testing of the AI outputs, is crucial to maintain safety (we don't want the AI suggesting anything non-compliant or making actual transactions).

MCP Integration: If the organisation is deploying MCP servers (Model Context Protocol) to expose certain backend capabilities to AI, our assistant can leverage them in a controlled way. MCP provides a standardized API for tools and data that AI agents can use [14]. For example, if there is an MCP server for checking current market data, the assistant (with the user's permission) might call it to incorporate real-time info into its answer. However, given this is an internal dev assistant, its primary use of MCP would likely be to access *internal knowledge bases or tools* (like a code search tool) rather than end-user functions. We will ensure any such usage complies with security (the assistant will only call MCP endpoints that are read-only or otherwise safe, and only when it improves the help provided to the developer).

In essence, the AI assistant transforms the developer experience: from coding with just static documentation to having an interactive, knowledgeable mentor that knows your company's systems. This can greatly speed up development (by providing boilerplate code, catching mistakes early, and answering questions instantly) [13] [15], while also enforcing consistency and best practices across teams.

Deployment and Scaling Considerations

Platform Deployment: The platform itself will be deployed on the organisation's infrastructure, likely as a set of containerized services. A common approach is to use Kubernetes for the platform services too. We would create deployments for the IDE backend service, the AI service, maybe a gateway service, etc., within an internal cluster (separate from the user workload cluster for dev containers, or logically separated by namespace). This provides scalability and high availability. For instance, multiple instances of the IDE backend can run behind a load balancer to support many concurrent users. We'll employ standard HA techniques: load-balanced stateless web servers, and perhaps a small database for user/project metadata (which would be a replicated SQL or NoSQL store). All stateful components (like the code storage or vector DB for AI) should be set up with redundancy (e.g. the object storage cluster or NFS is highly available; the vector DB is replicated). The platform will also be deployed across multiple availability zones or data centers if required, to ensure continuity in case of outages.

Scaling Developer Environments: The Kubernetes cluster that runs user dev containers must scale out to handle peak loads (e.g. many developers working simultaneously). We will enable cluster auto-scaling (if on cloud) or have enough node capacity on-prem with the possibility to add nodes quickly. Each container is relatively lightweight (perhaps 1-2 vCPU and a few GB of RAM reserved by default), and we can over-provision to some extent, but we must monitor resource usage. Implementing the idle shutdown (as mentioned) helps recycle resources [5] – if a user hasn't been active for e.g. 30 minutes, their container stops, but their data remains saved. When they come back, a new container spins up with their volume mounted, restoring their workspace. This optimises resource usage so we can serve more users than active machines at a given time.

For scaling, we also plan for **horizontal scaling** of the AI assistant. If many users are invoking the AI simultaneously, we might need multiple AI worker instances or a GPU cluster. We can deploy, for example, several replicas of a GPU-serving pod each running the model, or use a model server that supports concurrency. Since LLM inference can be heavy, we'll monitor response times and possibly restrict the length/frequency of queries to manage load. Another approach is using a smaller model for quick autocomplete and reserving a larger model for full chat answers, balancing quality vs performance.

Deployment of Applications (DevOps Pipeline): The platform should integrate with the organisation's deployment targets. Likely, the bank has staging and production Kubernetes clusters or other hosting

for applications. Our platform will either directly deploy to those (via API) or integrate with CI/CD tools that do the deployment. A recommended model is **GitOps**: when code is ready, the developer's changes are merged into a git repository (maybe a separate release repo or branch), and a tool like Argo CD or Jenkins X picks that up and deploys to the cluster. However, since the prompt suggests the platform includes approval mechanisms, it could have a built-in "Promote to Environment" feature: e.g. a developer clicks *Deploy to Staging*, the platform builds a container image for the app (using Docker or a build service) and deploys it to a staging namespace. This could be done using Kubernetes under the hood (the platform might itself create a new Deployment in the target cluster for the app). For production, as described, it would pause for approval. We'll ensure this process is smooth: provide a UI for approvers to see diffs and test results, and a one-click approve that triggers actual deployment.

Granular Access & Environment Setup: Based on roles, not all environments are accessible to all users. We might have multiple Kubernetes clusters or namespaces: e.g. a **Dev cluster** for ephemeral test deployments (where any developer can deploy freely), a **QA/Staging** environment (where deployment might require lead approval), and a **Prod** environment (strictly controlled). The platform should be configured with the credentials to deploy to these clusters but will gate who can trigger what. This separation also aids scaling – prod cluster is separate and not impacted by heavy dev workloads.

Monitoring and Maintenance: Operating this platform at scale requires monitoring of system health. We will include dashboards for container resource usage, active user count, etc., and alerts when capacity is low or if any service fails. If Kubernetes is used, leverage Prometheus/Grafana for metrics. Monitor the AI assistant performance and usage as well (possibly logging how often suggestions are used, to further refine the model). Regular maintenance windows might be scheduled for upgrades (e.g. updating base images for security, or updating the LLM with new knowledge).

Disaster Recovery: In a regulated environment, we need backup strategies. All persistent data (code storage, databases, etc.) should be backed up on a regular schedule. The platform's infrastructure as code (Helm charts, manifests) should be stored in Git so that it can be rebuilt in a disaster scenario. If possible, use multiple data centers: e.g. primary and secondary site, where secondary can take over if primary fails, to minimise downtime (RTO/RPO considerations might apply as per internal policy).

User Scaling and Onboarding: The solution should handle potentially hundreds or thousands of developers. We will automate onboarding by linking to the corporate directory – as soon as a user is in the appropriate AD group, they can log in and a default development space is provisioned for them. To manage storage, quotas can be applied (e.g. each user can use X GB of storage unless extended). If demand grows, we plan capacity accordingly (add more K8s nodes or clusters, more GPU servers for AI, etc.). Scaling should be mostly transparent: Kubernetes and autoscaling policies will handle bursts by queuing container starts until resources free up or new nodes come online.

Diagrams/Tables: (See the architecture diagram above for the overall design.) If useful, we can tabulate some tech choices:

Aspect	Technology/Approach	Notes
IDE Framework	**Eclipse Theia** (browser IDE)	Extensible, VS Code-compatible, Monaco-based UI
Editor Engine	**Monaco Editor**	Rich code editing (from VS Code) 6
Orchestration	**Kubernetes**	Manage isolated dev containers 1

Aspect	Technology/Approach	Notes
File Storage	**S3/Object Storage** or PV	Persistent code storage mounted in containers [1]
Authentication	**OAuth2/OIDC + LDAP**	SSO via corp IdP, LDAP for roles [7]
AI Model	**Code LLM (on-prem)**	e.g. CodeLlama 34B fine-tuned on internal data
Knowledge Base	**Vector DB + RAG**	Embed internal docs for AI context [11]
Networking	**Ingress Controller**	Expose web app previews per user [4]
CI/CD Integration	**GitOps / ArgoCD or Internal Pipeline**	Controlled deployments with approvals
Security Scanning	**Static Analysis + AI review**	SonarQube/Codacy and AI validate code [15]

This combination of architecture and tools will achieve a robust, secure, and user-friendly platform for internal development. It marries the agility of a cloud IDE (fast setup, no local config needed) with the stringent security and compliance requirements of a financial institution. Developers will be able to rapidly prototype Python scripts or Next.js frontends in their browser, get AI-assisted help that knows their internal environment, and smoothly push applications through to production with proper oversight. By leveraging modern IDP (Internal Developer Platform) principles and AI augmentation, the organisation can accelerate development while maintaining full control over software quality and security.

Sources:

1. Kowshickraj, "Building DevStash — A Replit like Platform," *Medium*, Feb. 2025 – Describes an architecture using Kubernetes per-user containers, persistent S3 storage, and WebSockets for a Replit-style IDE [1] [3].
2. Together AI Blog, "Building a personalized code assistant with RAG Fine-tuning," June 2024 – Explains retrieval-augmented generation (RAG) for grounding LLMs on up-to-date internal documents [11] [10].
3. AWS Machine Learning Blog, "Harness the power of MCP servers with Amazon Bedrock Agents," Apr. 2025 – Introduces the Model Context Protocol (MCP) as a standard to connect LLMs to data/tools, using client-server architecture [14] [9].
4. Swimm.io, "Replit Ghostwriter vs. Copilot: 5 Key Differences," *Swimm Blog*, accessed 2025 – Highlights Ghostwriter's in-IDE AI features like code context awareness and proactive bug detection [8].
5. Spacelift.io, "20 Best AI-Powered Coding Assistant Tools in 2025," Oct. 2025 – Notes that Replit's Ghostwriter provides real-time code suggestions and debugging help, and that tools like Codiga combine AI suggestions with static analysis for best-practice enforcement [13] [15].
6. Keycloak Server Admin Guide, Red Hat (latest) – Discusses integration of OAuth2 with existing LDAP/AD user directories for enterprise SSO [7].

---

[1] [2] [3] [4] [5] [6] Building DevStash — A Replit like Platform | by Kowshickraj | Medium
https://medium.com/@kowshickraj21.2005/building-devstash-a-replit-like-platform-5174eef4c7f3

7 Server Administration Guide
https://www.keycloak.org/docs/latest/server_admin/index.html

8 12 Replit Ghostwriter vs. Copilot: 5 Differences & How to Choose
https://swimm.io/learn/ai-tools-for-developers/replit-ghostwriter-vs-copilot-5-key-differences-and-how-to-choose

9 14 Harness the power of MCP servers with Amazon Bedrock Agents | AWS Machine Learning Blog
https://aws.amazon.com/blogs/machine-learning/harness-the-power-of-mcp-servers-with-amazon-bedrock-agents/

10 11 Building a personalized code assistant with open-source LLMs using RAG Fine-tuning
https://www.together.ai/blog/rag-fine-tuning

13 15 20 Best AI-Powered Coding Assistant Tools in 2025
https://spacelift.io/blog/ai-coding-assistant-tools