이 전제(금융권 VDE, **로컬 PC에 VS Code IDE 설치/사용 불가**)에서는 "VS Code Server를 띄우고 로컬 VS Code 클라이언트로 붙는" 모델이 성립하지 않습니다. 대신 **브라우저만으로 개발**이 가능한 형태로 설계해야 합니다. 즉, 목표는 다음 중 하나입니다.

- **브라우저 기반 VS Code 환경(권장)**: code-server 또는 OpenVSCode Server를 내부망에 구축하고, VDE 단말은 브라우저로 접속
- (대안) **VDE 내부(가상 데스크톱)에서만 클라이언트 사용**: 만약 VDE 세션 내부에서 VS Code 설치가 허용된다면, 그 안에서 일반 원격 개발(SSH 등) 가능. 하지만 질문 취지상 "클라이언트 사용 불가"로 보고 제외하겠습니다.

아래는 금융권 VDE에서 실제로 통과 가능한 형태로, **브라우저 기반 "VS Code 서버(웹 IDE)" 구축 가이드**입니다.

---

1) 권장 목표 아키텍처("브라우저 IDE + 워크스페이스 격리 + 중앙감사")

구성(존/역할)

1. **VDE 단말(클라이언트)**
    - 브라우저만 사용 (Edge/Chrome)
    - 파일 다운로드/업로드, 클립보드, 프린트 등은 VDE 정책에 의해 통제
2. **접속 구간(보안 게이트)**
    - WAF/Reverse Proxy(Nginx) + TLS(사내 PKI)
    - SSO(OIDC/SAML) + MFA
    - 세션 타임아웃, IP/단말 정책, 접근제어
3. **IDE 존**
    - code-server 또는 OpenVSCode Server
    - 사용자별 격리된 워크스페이스(가장 중요)
    - 확장(extensions) 공급망(오프라인/사내 승인된 것만)
4. **소스/아티팩트 존**
    - 내부 Git(예: GitLab/Gitea/Bitbucket on-prem)
    - 내부 패키지 레지스트리(Nexus/Artifactory/Harbor)

- 언어별 미러(PyPI/NPM/Maven) — 폐쇄망이면 사실상 필수

5. **빌드/테스트 존(권장 분리)**
   - IDE 서버에서 직접 빌드·테스트를 수행하지 않고,
   - CI Runner/빌드 노드에서 수행 (리소스/감사/악성코드 리스크 분산)
6. **감사/로그 존**
   - 프록시 접속로그, 인증로그, 서버 감사(auditd), sudo 로그, 애플리케이션 로그를 중앙 수집(SIEM)

---

2) 핵심 설계 결정: "멀티테넌트 vs 사용자 전용 인스턴스"

금융권 VDE에서 가장 자주 문제가 되는 지점이 **사용자 간 격리**입니다.

A안(권장): "사용자 전용 IDE 인스턴스" (1인 1컨테이너/1VM)

- 장점: 격리 명확, 권한/자원/로그 분리 쉬움, 감사 대응 용이
- 단점: 운영 리소스가 더 듦(오케스트레이션 필요)

B안(가능하지만 주의): "하나의 IDE 서버 + 리눅스 계정 분리"

- 장점: 단순
- 단점: 운영 중 권한 경계가 무너질 위험이 커서 금융권 심사에서 까다로울 수 있음
  (특히 확장 설치, 홈 디렉터리, 프로세스 권한, 소켓 접근 등)

결론: 가급적 **A안(사용자 전용 인스턴스)** 으로 설계하십시오.

---

3) 제품 선택: code-server vs OpenVSCode Server

1순위 권장: code-server

- 브라우저 IDE 구현 및 운영 옵션이 명확하고, 에어갭/사내 운영 패턴이 많습니다.
- 리버스 프록시+SSO 앞단 통제가 쉬운 편입니다.

대안: OpenVSCode Server

- "웹에서 VS Code 경험"에 초점을 둔 선택지

- 다만 실무 통제(확장/정책/로깅)는 어차피 별도 설계가 필요합니다.

※ 둘 중 무엇을 쓰든, 금융권 VDE에서 중요한 건 "도구 자체"보다 **접근통제·격리·감사·공급망**입니다.

---

4) 구축 절차(권장안): "브라우저 IDE"를 안전하게 올리는 순서

Step 0. 전제 확인(정책/통제)

- VDE에서 외부 인터넷 차단 여부(대부분 차단)
- VDE에서 브라우저로 내부 443 접근 가능 여부
- 파일 반출입 정책(업/다운로드 금지 여부)
- 클립보드/프린트/드라이브 매핑 정책
- 개발자가 사용할 언어/프레임워크(패키지 미러 필요 범위 결정)

Step 1. 내부 PKI 기반 TLS 종단(Reverse Proxy)

- 외부 노출은 443만
- IDE 프로세스는 localhost 바인딩(내부적으로만 동작)
- 강제 HTTPS, 최신 TLS 정책 적용

Step 2. SSO + MFA (필수)

- 권장 패턴: **프록시 계층에서 인증을 끝내고** IDE는 신뢰된 헤더/세션만 받게 구성
- OIDC/SAML (Keycloak/ADFS/Okta on-prem 등) 연동
- 그룹/조직 단위로 워크스페이스 접근 제어

Step 3. 사용자별 격리된 워크스페이스

- A안일 경우
    - 사용자 로그인 시: 사용자 전용 컨테이너/VM 생성 → code-server 기동 → 라우팅
    - 자원 제한: CPU/RAM/Disk quota
- 파일 저장소:
    - 개인 영구 볼륨(홈 디렉터리) + 프로젝트 체크아웃 볼륨 분리 권장

Step 4. "오프라인 확장(Extensions) 운영"

- **VSIX 화이트리스트**(승인된 확장만)
- 확장 자동 업데이트 금지(또는 중앙 통제)
- 내부 확장 저장소(사내 아티팩트)로만 설치되게 유도

Step 5. 언어 패키지 미러(거의 필수)

- Python: 내부 PyPI 미러/프록시
- Node: 내부 NPM 프록시
- Java: Maven/Gradle 리포 미러
- OS 패키지: 내부 apt repo(또는 golden image에 포함)

Step 6. 감사/로그 수집

최소:

- SSO 인증 이벤트(성공/실패)
- 프록시 접근로그(사용자/세션/IP/URL/시간)
- 서버 관리자 행위(sudo/auditd)
  권장:
- IDE 서비스 로그(세션 시작/종료, 에러)
- "터미널 명령 전문"은 조직 정책에 따라 민감할 수 있으므로, 보통은 **실행 이벤트/프로세스 수준 + 중요한 변경 이벤트**로 절충합니다.

---

5) VDE 환경에서 자주 걸리는 "실무 이슈"와 해결 방향

이슈 1) 브라우저 IDE에서 파일 업/다운로드가 막힘

- 해결:
  - 개발은 내부 Git로만(커밋/푸시가 곧 저장/반출입)
  - 산출물은 아티팩트 저장소에 업로드
  - 외부 반출은 별도 반출승인 프로세스(보안팀 워크플로우)로 처리

이슈 2) 확장 설치가 제멋대로 되면 공급망 통제가 깨짐

- 해결:
    - "확장 설치 권한"을 중앙에서만 허용
    - 사내 승인 VSIX만 제공
    - 확장 설치 이력도 감사 로그로 남김

이슈 3) 멀티유저 단일 서버에서 권한 경계 문제가 발생

- 해결:
    - 1인 1컨테이너/VM로 전환
    - 최소한 리눅스 계정 + cgroup + mount namespace + 권한 분리

이슈 4) 성능(특히 대규모 monorepo)

- 해결:
    - 워크스페이스 디스크는 NVMe급 스토리지
    - 인덱싱/언어서버 리소스 상향
    - 빌드는 CI로 분리(IDE 서버는 편집/테스트 일부만)

---

6) 추천 "표준 배포 모델" 2가지

모델 A(금융권 심사 친화적, 권장)

- **Kubernetes(또는 Nomad) + 1인 1컨테이너 IDE**
- Ingress(Nginx) + OIDC
- 사용자 볼륨(PVC) + 내부 Git/레지스트리
- 중앙 로깅/모니터링

모델 B(작게 시작, 이후 확장)

- **VM 기반 1인 1VM IDE**
- 프록시에서 SSO
- VM 템플릿(골든이미지)로 표준화
- 사용자 증가 시 오케스트레이션 도입

---

7) 다음 단계로 제가 바로 만들어 드릴 수 있는 것(요청만 주시면 됨)

귀하의 전제가 이미 명확하므로, 추가 질문 없이도 "금융권 VDE 표준안"으로 아래를 즉시 구성해 드릴 수 있습니다.

- 네트워크/존 구성도 + 포트/통신 매트릭스
- 보안통제 체크리스트(접근통제/인증/감사/공급망/취약점 대응)
- 운영 절차서(확장 반입, 이미지 업데이트, 계정/권한, 로그 보관)
- 초기 PoC용 배포 스펙(동시 사용자 수 가정: 예 20/50/100)별 사이징 가이드

원하시는 방향을 제가 "기본값"으로 잡아 진행하겠습니다:

- **code-server**
- **1인 1컨테이너(또는 1인 1VM)**
- **프록시에서 OIDC SSO+MFA**
- **VSIX 화이트리스트 + 내부 패키지 미러**
- **빌드/테스트는 CI 존 분리**

이 구성으로 진행하면 "로컬 PC에 IDE 설치 불가" 조건에서도 VDE에서 브라우저만으로 개발이 가능하며, 금융권 필수 통제(격리/감사/공급망)를 구조적으로 충족시키기 좋습니다.

전제(VDE 환경, 로컬 PC에 VS Code IDE 불가)까지 포함하면 "Cursor처럼"을 **브라우저 기반 IDE + 사내(폐쇄망) AI 코딩 백엔드**로 재구성해야 합니다. 핵심은 다음 3가지를 동시에 만족시키는 것입니다.

1. **브라우저만으로 개발(웹 IDE)**
2. **Copilot/Cursor 급 사용자 경험(자동완성 + 채팅/편집 + 에이전트)**
3. **금융권 수준의 통제(격리·감사·공급망·데이터 비유출)**

아래는 그 기준에서의 **구축 방안(권장 아키텍처 + 구현 로드맵 + 운영통제)** 입니다.

---

1) 전체 아키텍처(권장) — "Web IDE + AI Gateway + Code Intelligence"

1. Web IDE 계층

- **code-server(브라우저 VS Code)** 를 사용자에게 제공
  - VDE 단말은 브라우저로만 접속
  - code-server는 "웹에서 VS Code 경험"을 제공하는 대표적인 OSS 선택지입니다. Visual Studio Marketplace
- (대안) OpenVSCode Server도 가능하나, 실운영 통제는 어차피 별도 설계가 필요합니다. GitHub

2. AI 코딩 기능 계층("Cursor 경험"을 2트랙으로 분리)

Cursor류 기능은 크게 두 가지입니다.

A) 인라인 자동완성(Copilot 스타일)

- **Tabby**: 사내 구축 가능한 "self-hosted AI coding assistant"로, Copilot 대안으로 명시되어 있습니다. GitHub+1
- 이 트랙은 *지연시간(Latency)* 이 핵심이므로, 별도 서버로 최적화하는 편이 안정적입니다.

B) 채팅/편집/리팩토링/에이전트(멀티파일 수정, 작업 수행)

- **Continue**: VS Code 계열 IDE 확장으로 "에이전트/챗/편집" 워크플로우를 제공하는 OSS입니다. Visual Studio Marketplace+1
- code-server는 "VS Code 기반"이므로, Continue 같은 확장을 **웹 IDE 내부에 설치**

하는 방식으로 동일 UX를 구현하는 것이 현실적인 경로입니다.

3. Code Intelligence / RAG 계층(선택이지만 "Cursor급"을 원하면 사실상 필수)

- 조직 코드베이스가 크면 "코드 검색/심볼/레퍼런스/크로스레포 컨텍스트"가 성능을 좌우합니다.
- 엔터프라이즈에서는 Sourcegraph + Cody 같은 조합이 자주 쓰이며, self-hosted 옵션도 언급됩니다. sourcegraph.com+1
- 단, 라이선스/도입정책이 맞지 않으면 내부 코드 인덱서(리포 스캔 + 임베딩/RAG)로 자체 구현해도 됩니다.

4. LLM Serving 계층

- 내부 GPU 팜(H100/A100/RTX 5090 등) 위에 **vLLM/TGI/TensorRT-LLM** 중 하나로 "사내 LLM 엔드포인트" 제공
- 여기에 정책/감사/차단을 걸기 위해 **LLM Gateway**(사내 API) 레이어를 두는 것을 권장:
  - 모델 라우팅(autocomplete vs chat)
  - 프롬프트/응답 로깅 정책(보존/마스킹/필터)
  - 토큰/요금(사내 과금) / rate limit
  - DLP 룰(민감정보 탐지)

---

2) 배포 모델 — 금융권 VDE에서 "통과 가능한" 표준안

모델 A(권장): "1인 1 IDE 인스턴스" + 중앙 AI 백엔드

- 접속: VDE 브라우저 → Reverse Proxy(SSO/MFA) → 사용자별 code-server 컨테이너/VM
- 장점: **격리**가 명확하여 심사/감사에 유리(가장 중요)
- AI: Tabby 서버(공유) + LLM Gateway(공유) + (선택) Code RAG(공유)

모델 B(비권장에 가깝지만 가능): "공용 IDE 서버" + OS 계정 격리

- 운영은 쉬우나, 확장 설치/권한/프로세스 경계가 깨지기 쉬워 금융권에서는 리스크가 큽니다.

---

3) "Cursor처럼"을 기능 단위로 매핑한 구현 체크리스트

(1) 인라인 자동완성(필수)

- Tabby 서버 구축 (사내망)
- code-server에 Tabby 클라이언트/확장 설치(또는 Tabby가 제공하는 IDE 연동 방식)
- 성능 목표:
    - P95 150~250ms 내외(가능하면)로 맞춰야 "쓸만함" 평가를 받습니다.

Tabby는 "self-hosted AI coding assistant"로 명시되어 있고, 팀이 자체 LLM 기반 자동완성 서버를 구축할 수 있음을 설명합니다. tabby.tabbyml.com+1

(2) Chat + Edit + Agent(필수)

- Continue 확장(웹 IDE 내부 설치) + 사내 LLM Gateway 연결
- Continue는 VS Code 확장으로 에이전트/챗/편집 기능을 제공하는 방향을 명시합니다. Visual Studio Marketplace+1

(3) Repo-aware 컨텍스트(권장)

- 최소: 현재 워크스페이스 파일 + git diff + 최근 열람 파일
- 권장: 크로스레포 RAG("조직 코드 전체" 질의/요약/탐색)
    - Sourcegraph 계열은 self-hosted 보안 옵션을 강조합니다. sourcegraph.com+1

(4) PR/이슈 연동(선택)

- 내부 Git(대부분 GitHub Enterprise Server / GitLab)과의 연동은 "에이전트가 PR 생성/수정"까지 가려면 필요합니다.
- 단, GitHub Copilot류는 GitHub.com 계정/클라우드 의존성이 있어 완전 폐쇄망과 충돌하는 경우가 흔합니다. GitHub+1

---

4) 구축 로드맵(현실적인 단계)

Phase 1 — PoC(2~4주)

목표: "정말 Cursor처럼 체감되는가"를 검증

1. code-server 1인 1컨테이너 모델로 접속 성공(SSO는 간소화 가능)
2. Tabby 서버 + 소형 코드 모델로 인라인 자동완성 체감 확인
3. Continue + 내부 LLM(예: 코드 특화 모델)로 Chat/Edit 수행
4. 로그/감사 최소 요건(접속/세션/관리자행위) 중앙 수집

Phase 2 — Pilot(4~8주)

목표: "금융권 통제"를 충족하는 운영형으로 전환

1. SSO(MFA) + 세션정책 + IP allowlist
2. 확장(Extensions) **VSIX 화이트리스트** + 내부 배포(공급망 통제)
3. 내부 패키지 미러(PyPI/NPM/Maven/apt)
4. LLM Gateway 도입(레이트리밋, 모델 라우팅, 로깅/마스킹, 정책)
5. (선택) 코드 인덱싱/RAG 구축

Phase 3 — Production

목표: 대규모 사용자/대규모 코드베이스 운영

- 오케스트레이션(K8s/Nomad)로 IDE 인스턴스 자동 프로비저닝
- 모델 Serving 고도화(vLLM 등) + GPU 스케줄링
- 품질 측정(수용률, 지연, 정확도, 보안 이벤트) 대시보드화
- 모델/프롬프트 변경관리(승인 프로세스, A/B, 롤백)

---

5) 보안·감사·데이터 통제(금융권 핵심 포인트)

1. **데이터 비유출**

- IDE → LLM은 사내 Gateway로만 통신(인터넷 완전 차단)
- 프롬프트/응답 보존 정책(예: 기본 미보존, 사건 대응용 최소 로그만)

2. **격리**

- 사용자별 IDE 인스턴스(컨테이너/VM)

- 워크스페이스 스토리지 분리 + quota
- 빌드/테스트는 가능하면 별도 존(CI)으로 분리

3. **공급망**

- 확장(VSIX), 모델 파일, 컨테이너 이미지, 패키지 모두 "반입 승인" 체계
- 해시 고정(SHA256) + SBOM(가능하면)로 변경 추적

4. **감사**

- SSO 이벤트, 프록시 접근로그, IDE 세션 수명주기, 관리자 sudo/auditd
- (조직 정책 허용 시) 터미널 커맨드 감사는 매우 민감하므로 단계적으로 적용

---

6) 추천 기술 스택("즉시 실행 가능한 조합")

- Web IDE: **code-server** Visual Studio Marketplace
- Autocomplete: **Tabby**(self-hosted) GitHub+1
- Chat/Edit/Agent: **Continue 확장** Visual Studio Marketplace+1
- Code Intelligence(선택): Sourcegraph/Cody 또는 자체 RAG sourcegraph.com+1
- LLM Serving: vLLM/TGI/TensorRT-LLM + 사내 LLM Gateway(정책/감사/라우팅)

---

7) 다음 산출물(제가 바로 작성 가능)

원하시면, 위 구성을 "SI 제출/내부 심사"에 바로 쓰도록 다음 문서 세트를 구성해 드릴 수 있습니다.

1. **구축 아키텍처 설계서**(존 구성도, 데이터흐름, 포트 매트릭스)
2. **보안 통제 명세서**(SSO/MFA, 격리, 로깅, 공급망, 반입절차)
3. **운영 가이드**(확장/모델 업데이트, 장애/성능 대응, 계정/권한)
4. **PoC→Pilot→운영 로드맵 & 비용/자원 산정(사용자 수 기준)**

귀하 환경(이미 언급하신 H100/A100/5090 및 일반 서버)이라면, 추가 장비 도입 없이도 **"Autocomplete(저지연) + Chat/Edit(고성능)"** 를 분리 운영하여 체감 품질을 확보할 가능성이 높습니다.

원하시는 출력 형태가 "문서(설계서/명세서/운영가이드)"인지, 아니면 "구체 배포 스크립트(K8s 매니페스트/Helm, Nginx/OIDC 설정, Tabby/LLM gateway 구성)"까지 포함인지에 따라, 제가 다음 답변에서 바로 패키징 수준을 정해 제시하겠습니다.

전제(VDE 환경, 로컬 PC에 VS Code IDE 불가)까지 포함하면 "Cursor처럼"을 **브라우저 기반 IDE + 사내(폐쇄망) AI 코딩 백엔드**로 재구성해야 합니다. 핵심은 다음 3가지를 동시에 만족시키는 것입니다.

1. **브라우저만으로 개발(웹 IDE)**
2. **Copilot/Cursor 급 사용자 경험(자동완성 + 채팅/편집 + 에이전트)**
3. **금융권 수준의 통제(격리·감사·공급망·데이터 비유출)**

아래는 그 기준에서의 **구축 방안(권장 아키텍처 + 구현 로드맵 + 운영통제)** 입니다.

---

1) 전체 아키텍처(권장) — "Web IDE + AI Gateway + Code Intelligence"

1. Web IDE 계층

- **code-server(브라우저 VS Code)** 를 사용자에게 제공
  - VDE 단말은 브라우저로만 접속
  - code-server는 "웹에서 VS Code 경험"을 제공하는 대표적인 OSS 선택지입니다. Visual Studio Marketplace
- (대안) OpenVSCode Server도 가능하나, 실운영 통제는 어차피 별도 설계가 필요합니다. GitHub

2. AI 코딩 기능 계층("Cursor 경험"을 2트랙으로 분리)

Cursor류 기능은 크게 두 가지입니다.

A) 인라인 자동완성(Copilot 스타일)

- **Tabby**: 사내 구축 가능한 "self-hosted AI coding assistant"로, Copilot 대안으로 명시되어 있습니다. GitHub+1
- 이 트랙은 *지연시간(Latency)* 이 핵심이므로, 별도 서버로 최적화하는 편이 안정적입니다.

B) 채팅/편집/리팩토링/에이전트(멀티파일 수정, 작업 수행)

- **Continue**: VS Code 계열 IDE 확장으로 "에이전트/챗/편집" 워크플로우를 제공하는 OSS입니다. Visual Studio Marketplace+1
- code-server는 "VS Code 기반"이므로, Continue 같은 확장을 **웹 IDE 내부에 설치**

하는 방식으로 동일 UX를 구현하는 것이 현실적인 경로입니다.

3. Code Intelligence / RAG 계층(선택이지만 "Cursor급"을 원하면 사실상 필수)

- 조직 코드베이스가 크면 "코드 검색/심볼/레퍼런스/크로스레포 컨텍스트"가 성능을 좌우합니다.
- 엔터프라이즈에서는 Sourcegraph + Cody 같은 조합이 자주 쓰이며, self-hosted 옵션도 언급됩니다. sourcegraph.com+1
- 단, 라이선스/도입정책이 맞지 않으면 내부 코드 인덱서(리포 스캔 + 임베딩/RAG)로 자체 구현해도 됩니다.

4. LLM Serving 계층

- 내부 GPU 팜(H100/A100/RTX 5090 등) 위에 **vLLM/TGI/TensorRT-LLM** 중 하나로 "사내 LLM 엔드포인트" 제공
- 여기에 정책/감사/차단을 걸기 위해 **LLM Gateway**(사내 API) 레이어를 두는 것을 권장:
  - 모델 라우팅(autocomplete vs chat)
  - 프롬프트/응답 로깅 정책(보존/마스킹/필터)
  - 토큰/요금(사내 과금) / rate limit
  - DLP 룰(민감정보 탐지)

---

2) 배포 모델 — 금융권 VDE에서 "통과 가능한" 표준안

모델 A(권장): "1인 1 IDE 인스턴스" + 중앙 AI 백엔드

- 접속: VDE 브라우저 → Reverse Proxy(SSO/MFA) → 사용자별 code-server 컨테이너/VM
- 장점: **격리**가 명확하여 심사/감사에 유리(가장 중요)
- AI: Tabby 서버(공유) + LLM Gateway(공유) + (선택) Code RAG(공유)

모델 B(비권장에 가깝지만 가능): "공용 IDE 서버" + OS 계정 격리

- 운영은 쉬우나, 확장 설치/권한/프로세스 경계가 깨지기 쉬워 금융권에서는 리스크가 큽니다.

---

3) "Cursor처럼"을 기능 단위로 매핑한 구현 체크리스트

(1) 인라인 자동완성(필수)

- Tabby 서버 구축 (사내망)
- code-server에 Tabby 클라이언트/확장 설치(또는 Tabby가 제공하는 IDE 연동 방식)
- 성능 목표:
    - P95 150~250ms 내외(가능하면)로 맞춰야 "쓸만함" 평가를 받습니다.

Tabby는 "self-hosted AI coding assistant"로 명시되어 있고, 팀이 자체 LLM 기반 자동완성 서버를 구축할 수 있음을 설명합니다. tabby.tabbyml.com+1

(2) Chat + Edit + Agent(필수)

- Continue 확장(웹 IDE 내부 설치) + 사내 LLM Gateway 연결
- Continue는 VS Code 확장으로 에이전트/챗/편집 기능을 제공하는 방향을 명시합니다. Visual Studio Marketplace+1

(3) Repo-aware 컨텍스트(권장)

- 최소: 현재 워크스페이스 파일 + git diff + 최근 열람 파일
- 권장: 크로스레포 RAG("조직 코드 전체" 질의/요약/탐색)
    - Sourcegraph 계열은 self-hosted 보안 옵션을 강조합니다. sourcegraph.com+1

(4) PR/이슈 연동(선택)

- 내부 Git(대부분 GitHub Enterprise Server / GitLab)과의 연동은 "에이전트가 PR 생성/수정"까지 가려면 필요합니다.
- 단, GitHub Copilot류는 GitHub.com 계정/클라우드 의존성이 있어 완전 폐쇄망과 충돌하는 경우가 흔합니다. GitHub+1

---

4) 구축 로드맵(현실적인 단계)

Phase 1 — PoC(2~4주)

목표: "정말 Cursor처럼 체감되는가"를 검증

1. code-server 1인 1컨테이너 모델로 접속 성공(SSO는 간소화 가능)
2. Tabby 서버 + 소형 코드 모델로 인라인 자동완성 체감 확인
3. Continue + 내부 LLM(예: 코드 특화 모델)로 Chat/Edit 수행
4. 로그/감사 최소 요건(접속/세션/관리자행위) 중앙 수집

Phase 2 — Pilot(4~8주)

목표: "금융권 통제"를 충족하는 운영형으로 전환

1. SSO(MFA) + 세션정책 + IP allowlist
2. 확장(Extensions) **VSIX 화이트리스트** + 내부 배포(공급망 통제)
3. 내부 패키지 미러(PyPI/NPM/Maven/apt)
4. LLM Gateway 도입(레이트리밋, 모델 라우팅, 로깅/마스킹, 정책)
5. (선택) 코드 인덱싱/RAG 구축

Phase 3 — Production

목표: 대규모 사용자/대규모 코드베이스 운영

- 오케스트레이션(K8s/Nomad)로 IDE 인스턴스 자동 프로비저닝
- 모델 Serving 고도화(vLLM 등) + GPU 스케줄링
- 품질 측정(수용률, 지연, 정확도, 보안 이벤트) 대시보드화
- 모델/프롬프트 변경관리(승인 프로세스, A/B, 롤백)

---

5) 보안·감사·데이터 통제(금융권 핵심 포인트)

1. **데이터 비유출**

- IDE → LLM은 사내 Gateway로만 통신(인터넷 완전 차단)
- 프롬프트/응답 보존 정책(예: 기본 미보존, 사건 대응용 최소 로그만)

2. **격리**

- 사용자별 IDE 인스턴스(컨테이너/VM)

- 워크스페이스 스토리지 분리 + quota
- 빌드/테스트는 가능하면 별도 존(CI)으로 분리

3. **공급망**

- 확장(VSIX), 모델 파일, 컨테이너 이미지, 패키지 모두 "반입 승인" 체계
- 해시 고정(SHA256) + SBOM(가능하면)로 변경 추적

4. **감사**

- SSO 이벤트, 프록시 접근로그, IDE 세션 수명주기, 관리자 sudo/auditd
- (조직 정책 허용 시) 터미널 커맨드 감사는 매우 민감하므로 단계적으로 적용

---

6) 추천 기술 스택("즉시 실행 가능한 조합")

- Web IDE: **code-server** Visual Studio Marketplace
- Autocomplete: **Tabby**(self-hosted) GitHub+1
- Chat/Edit/Agent: **Continue 확장** Visual Studio Marketplace+1
- Code Intelligence(선택): Sourcegraph/Cody 또는 자체 RAG sourcegraph.com+1
- LLM Serving: vLLM/TGI/TensorRT-LLM + 사내 LLM Gateway(정책/감사/라우팅)

---

7) 다음 산출물(제가 바로 작성 가능)

원하시면, 위 구성을 "SI 제출/내부 심사"에 바로 쓰도록 다음 문서 세트를 구성해 드릴 수 있습니다.

1. **구축 아키텍처 설계서**(존 구성도, 데이터흐름, 포트 매트릭스)
2. **보안 통제 명세서**(SSO/MFA, 격리, 로깅, 공급망, 반입절차)
3. **운영 가이드**(확장/모델 업데이트, 장애/성능 대응, 계정/권한)
4. **PoC→Pilot→운영 로드맵 & 비용/자원 산정(사용자 수 기준)**

귀하 환경(이미 언급하신 H100/A100/5090 및 일반 서버)이라면, 추가 장비 도입 없이도 **"Autocomplete(저지연) + Chat/Edit(고성능)"** 를 분리 운영하여 체감 품질을 확보할 가능성이 높습니다.

원하시는 출력 형태가 "문서(설계서/명세서/운영가이드)"인지, 아니면 "구체 배포 스크립트(K8s 매니페스트/Helm, Nginx/OIDC 설정, Tabby/LLM gateway 구성)"까지 포함인지에 따라, 제가 다음 답변에서 바로 패키징 수준을 정해 제시하겠습니다.