Agile7 Code Platform (ACP)

Software Architecture Whitepaper

This whitepaper presents the Agile7 Code Platform, a scalable AI-driven enterprise open-source no-code platform that ensures product requirements-to-code alignment while maintaining software architecture models as the single source of truth.

Document Version: v1.0b

Date: September 2, 2025

Author: Max Meinhardt

Agile7

maxm@agile7.com

www.agile7.com

Executive Summary

Agile7 Code Platform (ACP), a proof-of-concept open-source platform by Max Meinhardt, Agile7's founder, transforms enterprise software development by enforcing consistent alignment between product requirements, system architecture, and development through automation. Using UML models (ACP v1.0a) and supporting scripts as the sole source of truth, it addresses scope creep, architectural drift, and misalignment, integrating Business Requirements Validation to embed alignment in the development process. The requirements alignment score ensures code alignment with the product requirements and a code alignment score ensures generated code remains unmodified, preventing manual changes that could be overwritten during regeneration, thus enforcing UML and scripts as authoritative. This AI-first approach, set to replace manual coding, generates diverse code types, including microservices, agentic systems, and more, surpassing no-code platforms and AI-assisted tools by maintaining precise alignment. A visual dashboard allows managers and product owners to track code alignment with requirements and use cases in real time, ensuring it consistently meets their configurable minimum thresholds.

Table of Contents

Executive Summary........ 2
1) Introduction........ 5
2) Project Overview and Requirements........ 5
    2.1) Project Overview........ 5
    2.2) Functional Requirements........ 5
    2.3) Non-Functional Requirements........ 6
3) Feasibility Analysis........ 7
    3.1) Technical Feasibility........ 7
    3.2) Operational Feasibility........ 9
    3.3) Economic Feasibility........ 10
    3.4) Switching Large Language Models........ 10
4) Functional Specification........ 10
    4.1) Architectural Components Overview........ 10
    4.2) Architecture Diagram........ 12
    12
    4.3) Sequence Diagram........ 12
    4.4) System Workflow........ 14
    4.5) Business Requirements Validation (BRV)........ 14
    4.6) Change Handling and Regeneration........ 17
    4.7) Subsystem Support........ 18
    4.8) Incremental Code Generation........ 18
    4.9) Deployment Workflow for Multiple Environments........ 20
    4.10) IDE Plugins........ 22
    4.11) Error Handling and Recovery........ 22
    4.12) Security........ 22
    4.13) LLM Switching Process........ 23
    4.13.1) DeepCode as the Default LLM........ 23
    4.13.2) Alternate LLMs........ 24
    4.13.3) Considerations........ 24
    4.13.4) Future-Proofing........ 24
    4.14) Sample UML and Scripting for Code Generation........ 25
5) Model Parser Release Roadmap........ 29
    Version 1.0: Initial Release (Alpha Stage – Foundational Modeling Support)........ 29
    Version 1.1: Enhanced Release (Beta Stage – Reduced Ambiguity and Expanded Inputs)........ 30
    Version 2.0: Mature Release (Release Candidate Stage – Comprehensive Parsing Support)........ 31
    Version 3.0: Enterprise Release (General Availability – Scalable and Secure Parsing)........ 32
Glossary........ 33
References........ 35
Appendix........ 36

A.1) ACP Architecture Diagram's PlantUML Code........36
........37
A.2) ACP Sequence Diagram's PlantUML Code........37
A.3) ACP Code-Generation Example UML Diagram........39
About Agile7........40
Document Revision History........40
Author Biography........40
Max's LinkedIn Account: https://www.linkedin.com/in/maxmeinhardt/........40

1) Introduction

Agile7 Code Platform (ACP), a proof-of-concept architected by Max Meinhardt, Agile7's founder, is designed to address challenges such as scope creep, architectural drift, and requirements misalignment. The platform enforces UML models and scripts as the sole source of truth, ensuring non-destructive regeneration via CI/CD pipelines. It integrates Business Requirements Validation (BRV) functionality to maintain and automate business-technical alignment, using the ACP web app for PRD (Product Requirements Document) input and UML/script editing, along with LLM-agnostic abstractions and ACP IDE plugins for engineers. Producing scalable and secure code for enterprise software projects, this whitepaper evaluates the platform's feasibility, detailing its AI-driven architecture, security, debugging, error handling, and BRV, supplemented with architectural diagrams and sample code.

2) Project Overview and Requirements

2.1) Project Overview

The ACP platform addresses software development challenges, such as scope creep, architectural drift, and manual coding errors, by automating code generation from Unified Modeling Language (UML) models and scripts. Leveraging AI-driven code generation, it ensures model-driven purity, LLM-agnostic adaptability, business-technical alignment, and scalable, secure code for enterprise industries, including microservices, MCP-compliant servers, and agentic systems. It streamlines development by maintaining UML as the sole source of truth, enabling non-destructive regeneration and requirements validation to align code with evolving business needs, and supports hybrid local/cloud deployment for flexibility.

2.2) Functional Requirements

1. **Enforce Model-Driven Purity**: Ensure all changes occur at the UML/script level with non-destructive regeneration, eliminating manual code edits via CI/CD pipeline gates and read-only deploys, using git diffs to detect changes and regenerate affected parts (Section 4.6).
2. **UML-Driven Code Generation**: Generate executable code (e.g., Java/Spring, Python/Flask) for microservices, MCP-compliant servers, agentic systems, and other types from PlantUML diagrams and scripts (e.g., Python, OCL).
3. **Agentic and Traditional Systems**: Support code generation for microservices, MCP-compliant servers, and agentic systems, orchestrated by LLM prompts for dynamic workflows, with options for traditional flows deployed to AWS ECS. The ACP Prompt-Based Build Engine

generates self-contained code, including REST/JSON endpoints and integration logic (e.g., Kafka, AWS S3), compatible with specified UML deployment diagrams.

4. **Multi-Environment Deployment**: Enable code generation for Development, Staging, and Production environments with environment-specific configurations.
5. **Debugging Support**: Provide ACP IDE plugins (e.g., for VS Code, IntelliJ) to map errors and breakpoints to UML/scripts for model-level fixes, with read-only BRV access.
6. **Third-Party Integration**: Integrate with external APIs (e.g., AWS S3, Kafka) via UML-defined interactions.
7. **Business Requirements Validation (BRV)**: Allow Product Managers to input requirements and use cases (e.g., from a PRD in JSON, YAML, or natural language) via the ACP web app with forms and a visual diagram editor, generating JSON with embedded PlantUML for validation. Engineers use the same web app or IDE plugins for UML/script editing and read-only PRD access. Compare PRDs against UML/script implementations via a Requirements Fidelity Score (RFS), validated manually (via web app/IDE dashboard) or automatically in the CI/CD pipeline.
8. **Subsystem Support**: Model systems as UML subsystems, generating code for microservices, MCP-compliant servers, or agentic systems, supporting both agentic and traditional flows for risk mitigation (Section 4.7).
9. **Incremental Code Generation**: Support multi-prompt code generation for large-scale systems, decomposing UML models into subsystems and orchestrating prompts for scalability (Section 4.12).

2.3) Non-Functional Requirements

1. **Performance**: Achieve code generation in ~10 seconds per code fragment, with CI/CD validation (including Code Fidelity Score and Requirements Fidelity Score, RFS) in ~1 minute per system component. Orchestration of agentic systems achieves <100ms latency per request. PRD storage in encrypted systems (e.g., AWS S3, databases) ensures fast access for BRV.
2. **Security**: PRDs and traceability matrices are stored in encrypted systems (e.g., AWS S3, databases), with access restricted to authorized components (e.g., Requirements Parser, BRV dashboard).
3. **Scalability**: Support complex enterprise systems with domain-specific UML extensions.

4. **Usability**: Enable engineers to model with PlantUML and scripts via the ACP web app or IDE plugins, with AI-assisted editors. Product Managers use the web app for non-technical PRD input/editing.
5. **Adaptability**: Support LLM-agnostic code generation, switching models in approximately <1 hour with minimal reconfiguration and semi-automated tuning.
6. **Traceability**: Ensure >90% (configurable %) business-technical alignment via Requirements Fidelity Score (RFS) in BRV, validating PRD requirements against UML/script implementations.

3) Feasibility Analysis

3.1) Technical Feasibility

Core Technology: The ACP Prompt-Based Build Engine, implemented as an MCP (Model Context Protocol) server, powered by a fine-tuned Large Language Model (LLM, e.g., DeepCode with LoRA), interprets UML models (e.g., PlantUML component/sequence diagrams) and scripts (e.g., Python, Object Constraint Language [OCL]) to generate executable code (e.g., Java/Spring, Python/Flask). LoRA fine-tuning ensures high-fidelity output, validated by a Code Fidelity Score (measuring Abstract Syntax Tree [AST] similarity between UML and code, targeting >90% (default) or configurable % match). This MCP server format provides standardized context provision and RAG support, orchestration of agentic workflows, LLM-agnostic adaptability, secure PRD handling, and consistency with generated outputs.

1. **Business Requirements Validation (BRV)**: A Requirements Parser (extending the Model Parser) processes PRD inputs (JSON, YAML, or natural language) using NLP, creating a traceability matrix. The ACP CFS Validator computes a Code Fidelity Score, and the ACP RFS Validator computes a Requirements Fidelity Score (RFS, targeting >90% semantic match) to compare PRD requirements to UML/scripts, ensuring traceability. The ACP web app and IDE plugins provide a BRV dashboard for manual review, and the CI/CD pipeline automates RFS checks, rejecting low-alignment implementations.
2. **Performance Metrics**: Code generation completes in ~10 seconds, with CI/CD validation (including RFS) in ~1 minute and orchestration adding <100ms latency per request, supporting enterprise-scale systems.
3. **LLM-Agnosticism**: Abstractions like LiteLLM enable switching between LLMs (e.g., GPT, DeepCode) via configuration-based model selection, with automated regeneration and testing mitigating risks from model updates. Switching requires updating API endpoints and

revalidating prompts, typically completed in <1 hour. For example, a prompt template might look like:

```
{
"prompt": "Given UML: {uml_diagram}, generate Java/Spring REST
controller with error handling for {endpoint}",
"type": "controller",
"inputs": ["uml_diagram", "endpoint"]
}
```

4. **Risk Analysis**: LLM hallucinations (e.g., incorrect code logic) have a low probability (~5% post-RAG mitigation) but are addressed via fallback templates and CI/CD validation. UML parsing errors (e.g., invalid syntax) occur in <2% of cases, resolved by engineer corrections guided by detailed error logs. PRD parsing ambiguity is mitigated by structured formats (e.g., JSON) and AI-assisted validation.

5. **Risk Mitigation for Code-Generating LLM Updates**: The platform pins LLM versions in the CI/CD pipeline (e.g., via LiteLLM configuration) to lock specific models during development or production, preventing unexpected behavior from updates. Determinism is enhanced by setting low-temperature parameters (e.g., temperature=0) in the ACP Prompt-Based Build Engine for code generation, minimizing output variability, with LoRA fine-tuning aligning outputs to domain-specific needs. Robust validation includes automated unit and integration tests in the CI/CD pipeline to verify functional correctness and PRD compliance (e.g., GDPR requirements) post-LLM updates. A guardrail in the ACP Prompt-Based Build Engine ensures generated code aligns with UML specifications by comparing language-agnostic code ASTs against language-agnostic UML ASTs, rejecting deviations to maintain the sole source of truth. These measures reduce regression risks and ensure Code Fidelity Score and RFS remain >90%.

6. **Standardized Prompt Design for LLM-Agnosticism**: The ACP Prompt-Based Build Engine uses a fixed set of standardized, version-controlled prompt templates, incorporating patterns like Chain-of-Thought (CoT) and few-shot learning. These templates are stored in a Git repository alongside UML models and scripts, using JSON or YAML formats with placeholders for UML, script, and PRD inputs. LiteLLM abstracts prompt execution, mapping templates to model-specific API calls while preserving the core prompt structure. Prompt validation is integrated into the CI/CD pipeline, with automated tests comparing generated code against expected ASTs to ensure consistency across LLMs. Few-shot examples (e.g., UML-to-code mappings) anchor outputs, reducing model-specific variability. Prompts are optimized for clarity using structured inputs (e.g., UML ASTs from the Model Parser), minimizing ambiguity. A prompt maintenance process allows engineers to refine templates when switching LLMs,

using test results to adjust CoT steps or examples, ensuring <1-hour reconfiguration. This approach supports the platform's LLM-agnostic design while ensuring reliable code generation.

7. **Security**: Eliminating manual code edits, using CI/CD pipelines with Code Fidelity Score, RFS, and static analysis (e.g., SonarQube), and securing PRD storage (e.g., AWS S3 with encryption) are technically viable, reducing vulnerabilities (e.g., injection flaws).
8. **Model-Driven Purity**: Maintaining UML as the sole source of truth is achievable using non-destructive regeneration and CI/CD validation, supported by existing tools like Jenkins and Git.
9. **Enforcing Model-Driven Changes**: Rejecting non-compliant code via CI/CD pipelines and using pre-commit hooks/AI-assisted UML editors is feasible with current DevOps and IDE technologies.
10. **Sole-Source-of-Truth for Complex Systems**: Modeling complex systems with UML and validating against PRD requirements is viable using domain-specific extensions, semantic validation, and AI-driven tools.
11. **Hybrid Delivery**: Supporting standalone applications, IDE plugins (e.g., for VS Code, IntelliJ with BRV dashboard), and cloud APIs (e.g., AWS-hosted) is technically possible using standard platforms (e.g., AWS ECS, Docker).
12. **Challenges**: LLM hallucinations, UML modeling complexity, and PRD parsing ambiguity are mitigated by Retrieval-Augmented Generation (RAG), AI-assisted editors, and structured PRD formats.

3.2) Operational Feasibility

- **Adoption**: Engineers can learn UML (e.g., PlantUML) and scripting, while Product Managers input PRD requirements via the ACP web app, easing the learning curve. Non-technical users (e.g., business analysts) can model high-level workflows and validate requirements via the BRV dashboard.
- **Integration**: Supporting 3rd-party APIs (e.g., AWS S3, Kafka) via UML component diagrams and RAG-augmented prompts is feasible with current API standards.
- **Debugging and Validation**: ACP IDE plugins for model-level/code-level debugging and BRV (mapping errors/requirements to UML/scripts) are viable using existing IDE frameworks, preserving no-code edits.

3.3) Economic Feasibility

- **Cost Savings**: Automation eliminates manual coding, and BRV reduces requirements misalignment, lowering development/maintenance costs. Non-destructive regeneration minimizes technical debt.
- **Scalability**: Supporting large-scale apps via microservices, MCP-compliant servers, and agentic systems is economically viable, using standard cloud platforms (e.g., AWS ECS).
- **Investment**: Initial costs for LLM fine-tuning, Requirements Parser, RFS implementation, and web-based app development are offset by consistent and reliable business-to-product alignment and productivity gains, leveraging efficient LoRA training and existing cloud services.

3.4) Switching Large Language Models

ACP's Prompt-Based Build Engine is designed to be LLM-agnostic, allowing seamless switching between Large Language Models (LLMs) to adapt to enterprise needs, performance requirements, or advancements in model capabilities. This section outlines the process for switching LLMs, exemplified using DeepCode (DeepSeek, e.g., DeepCode-33B) as the default model, with a target timeframe of approximately one hour, including API updates, prompt revalidation, and optional LoRA fine-tuning.

4) Functional Specification

The ACP platform automates software development through a model-driven, AI-first approach, with components working together to ensure model-driven purity, scalability, and traceability.

4.1) Architectural Components Overview

The ACP platform comprises interconnected components critical to its model-driven, AI-first approach for enterprise-scale software development:

- **Model Parser**: Interprets PlantUML diagrams (e.g., component, sequence) and scripts (e.g., Python, OCL), generating language-agnostic Abstract Syntax Trees (ASTs) to represent system architecture. It generates language-agnostic ASTs to reduce the complexity of stitching (Section 4.8) subsystems across different languages, although further analysis in this area is needed. It decomposes large UML models into subsystems with subsystem stereotypes, mapping UML elements to script functions for modular code generation (Sections 4.7, 4.14).
- **Requirements Parser**: Processes Product Requirements Document (PRD) inputs (JSON, YAML, or natural language) using NLP to extract requirements and use cases, creating a traceability matrix for Business Requirements Validation (BRV). It maps PRD requirements to

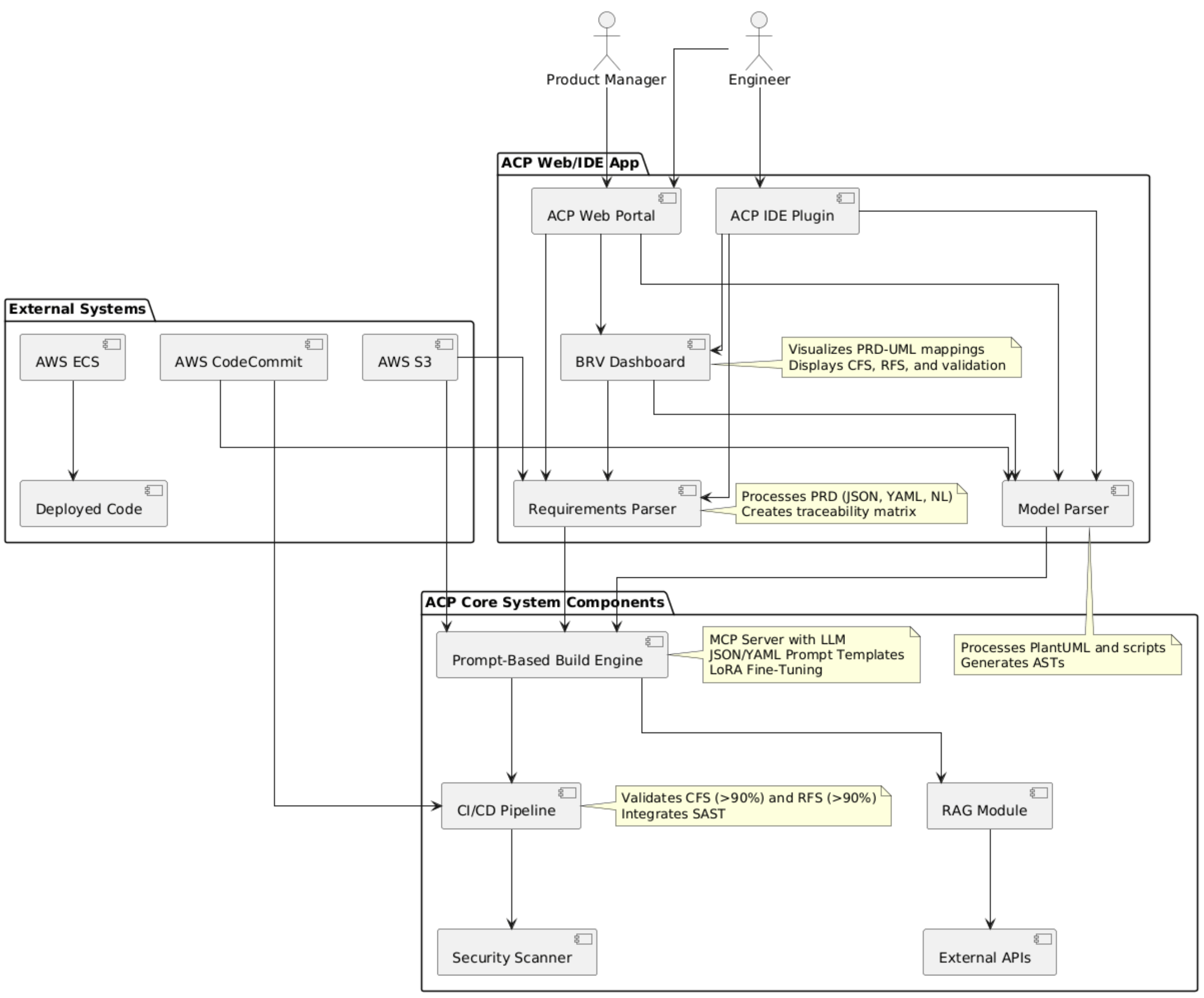
subsystem-specific UML elements, ensuring alignment via the Requirements Fidelity Score (RFS) (Section 4.5).

- **Prompt-Based Build Engine**: Implemented as an MCP (Model Context Protocol) server, this uses a fine-tuned Large Language Model (LLM, e.g., DeepCode with LoRA) with standardized, version-controlled prompt templates (JSON/YAML) to generate code (e.g., Java/Spring, Python/Flask). Augmented by Retrieval-Augmented Generation (RAG) for third-party APIs and libraries, it constructs and orchestrates multiple prompts per subsystem (e.g., for REST controllers, integration logic), stitching fragments into a cohesive codebase, often in parallel via AWS Lambda. Deployed as a cloud-hosted MCP server (e.g., on AWS ECS) with REST/JSON endpoints for prompt orchestration, integrating with the Model Parser and Requirements Parser as MCP-connected tools. It uses MCP's open standard to handle role-based access (e.g., read-only PRD views for engineers) and parallel execution, while pinning LLM versions for determinism.
- **CI/CD Pipeline**: Validates code fragments using the Code Fidelity Score (comparing code ASTs to UML/script ASTs), RFS (>90% PRD alignment), and security scans (e.g., SonarQube). It runs unit/integration tests with mock dependencies (e.g., Dockerized services) and deploys to AWS ECS. It supports targeted regeneration for updates (Section 4.6).
- **ACP Web Portal and IDE Plugin**: Enables engineers (Regular/Admin) to edit UML/scripts in a graphical editor, view read-only PRD data in BRV, and trigger “Show Mappings” (UML/code cross-referencing, real-time optional) and “Generate Code” (code generation). Managers view UML read-only, edit PRDs in BRV. Supports optional visual UML processing with LLaVA-1.5. Secured by OAuth 2.0/RBAC (Sections 4.5, 4.10).
- **BRV Dashboard**: Accessible via web app/IDE plugin, allows engineers (Regular/Admin) read-only PRD/UML access, Managers to edit PRDs, view UML read-only. Validates PRD alignment (RFS >90%) with traceability. Supports text-based and optional visual UML inputs (Section 4.5).
- **RAG Module**: Augments the LLM with external data (e.g., API documentation) for accurate third-party integration code, providing context for subsystem-specific prompts during multi-prompt generation.
- **Security Scanner (optional)**: Integrates SAST tools (e.g., SonarQube, Fortify) to execute OWASP-compliant scans in the CI/CD pipeline for compliance, scanning subsystem code fragments during incremental validation. (Section 4.12)

4.2) Architecture Diagram

The ACP system architecture diagram is shown as Figure 1 below. Its corresponding PlantUML code is in the Appendix Section A1 and can be viewed by copying and pasting it into the form at https://www.plantuml.com/plantuml/.

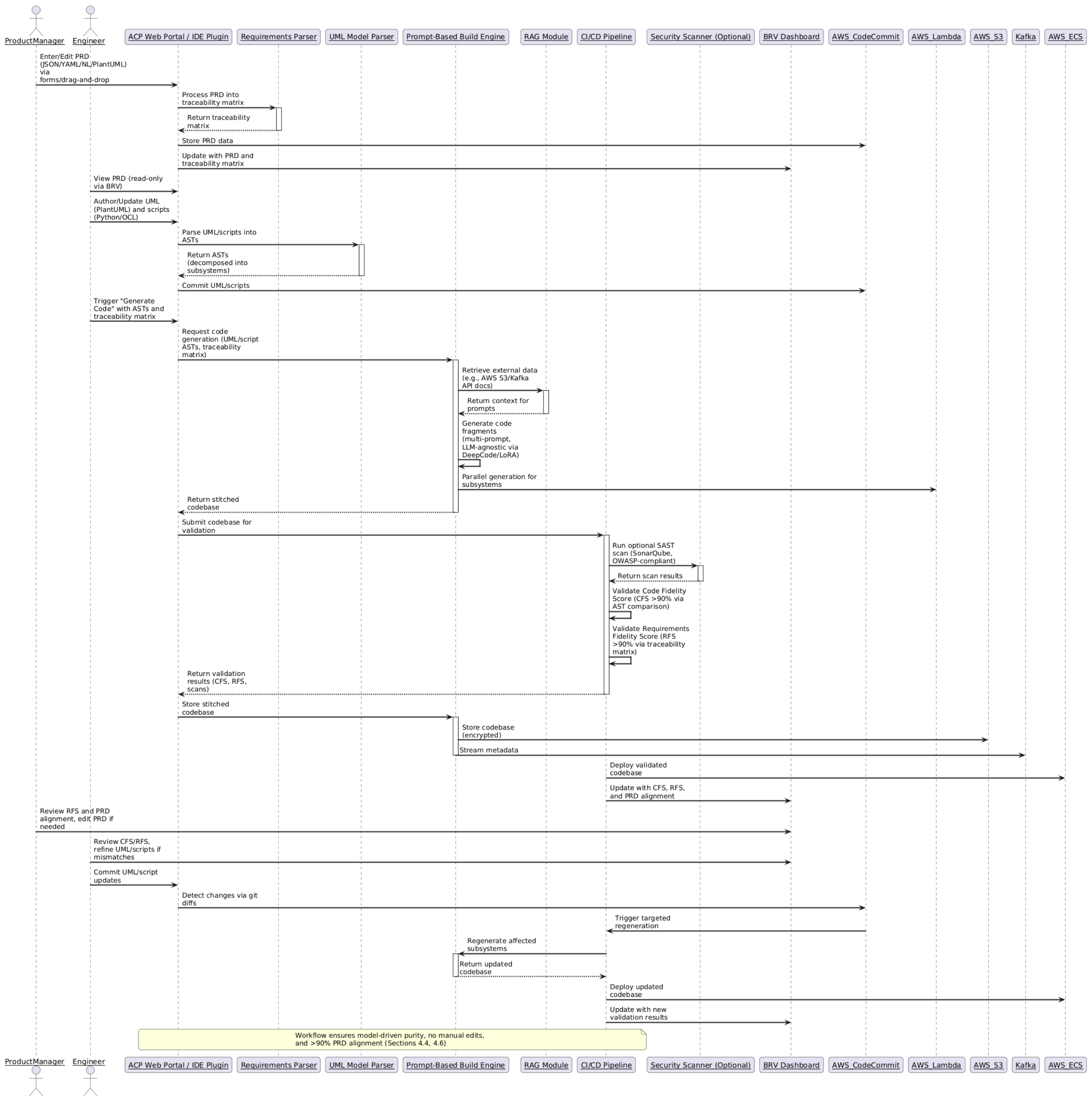
ACP Architecture Diagram (Figure 1)



4.3) Sequence Diagram

The ACP sequence diagram in Figure 2 below illustrates the incremental code generation and Business Requirements Validation (BRV) processes for the ACP platform. The diagram's associated PlantUML code is in the Appendix Section A2 and can be viewed by copying and pasting it into the form at https://www.plantuml.com/plantuml/.

ACP Sequence Diagram (Figure 2)


ProductManager
Engineer
ACP Web Portal / IDE Plugin
Requirements Parser
UML Model Parser
Prompt-Based Build Engine
RAG Module
CI/CD Pipeline
Security Scanner (Optional)
BRV Dashboard
AWS_CodeCommit
AWS_Lambda
AWS_S3
Kafka
AWS_ECS
Enter/Edit PRD (JSON/YAML/NL/PlantUML) via forms/drag-and-drop
Process PRD into traceability matrix
Return traceability matrix
Store PRD data
Update with PRD and traceability matrix
View PRD (read-only via BRV)
Author/Update UML (PlantUML) and scripts (Python/OCL)
Parse UML/scripts into ASTs
Return ASTs (decomposed into subsystems)
Commit UML/scripts
Trigger "Generate Code" with ASTs and traceability matrix
Request code generation (UML/script ASTs, traceability matrix)
Retrieve external data (e.g., AWS S3/Kafka API docs)
Return context for prompts
Generate code fragments (multi-prompt, LLM-agnostic via DeepCode/LoRA)
Parallel generation for subsystems
Return stitched codebase
Submit codebase for validation
Run optional SAST scan (SonarQube, OWASP-compliant)
Return scan results
Validate Code Fidelity Score (CFS >90% via AST comparison)
Validate Requirements Fidelity Score (RFS >90% via traceability matrix)
Return validation results (CFS, RFS, scans)
Store stitched codebase
Store codebase (encrypted)
Stream metadata
Deploy validated codebase
Update with CFS, RFS, and PRD alignment
Review RFS and PRD alignment, edit PRD if needed
Review CFS/RFS, refine UML/scripts if mismatches
Commit UML/script updates
Detect changes via git diffs
Trigger targeted regeneration
Regenerate affected subsystems
Return updated codebase
Deploy updated codebase
Update with new validation results
Workflow ensures model-driven purity, no manual edits, and >90% PRD alignment (Sections 4.4, 4.6)

4.4) System Workflow

1. **Input Processing**: Product Managers enter PRD (Product Requirements Document) data via the ACP web app's visual text and use-case creation tools, stored in AWS CodeCommit. The ACP Requirements Parser processes PRD data into a traceability matrix, while Engineers view PRD data in the web app or IDE plugins to create/update UML models (e.g., PlantUML diagrams) and scripts (e.g., Python, OCL). The Model Parser converts UML/scripts into ASTs, preprocessing visual UML into text for the LLM (default is DeepCode).
2. **Code Generation**: The Prompt-Based Build Engine uses DeepCode, fine-tuned via LoRA, to generate code fragments from UML/script ASTs and the PRD traceability matrix using JSON/YAML prompts. AWS Lambda enables parallel generation for subsystems, meeting ~10-second performance (Sections 4.8, 4.11).
3. **Validation**: The CI/CD Pipeline (Jenkins) validates fragments with the ACP CFS Validator (AST comparison, CFS >90% default) and RFS Validator (PRD alignment via traceability matrix, RFS >90% default). Engineers refine UML/scripts based on errors, while Product Managers verify RFS via the BRV Dashboard. Unit/integration tests and optional SonarQube scans run, with AWS Lambda supporting parallel validation (Sections 4.5, 4.11, 4.14).
4. **Stitching and Storage**: The Build Engine stitches fragments into a codebase, stored in AWS S3, with Kafka streaming metadata (Section 4.8).
5. **Deployment**: Validated code deploys to AWS ECS, with the BRV Dashboard enabling Product Managers and Engineers to monitor CFS, RFS, and PRD alignment (Section 4.9).
6. **Regeneration**: Engineers trigger targeted regeneration by committing UML/script updates to CodeCommit, using git diffs to identify changes. Alternatively, updates can be processed locally via the web app/IDE plugins, sending diffs to the CI/CD Pipeline via API/Kafka for prototyping, maintaining model-driven purity through validation. Committing ensures version control and traceability (Sections 4.8, 4.11).

4.5) Business Requirements Validation (BRV)

The BRV feature, implemented in the ACP web app hosted on AWS, enables Product Managers (admin/manager privileges) to input and edit Product Requirements Document (PRD) content, including standard elements such as functional requirements (e.g., "Process payments securely"), use cases (e.g., user-initiated payment workflows), and constraints (e.g., GDPR compliance), using JSON, YAML, natural language, or embedded PlantUML use-case diagrams via forms and a drag-and-drop editor. This ensures non-technical Product Managers can define business needs effectively, aligning with industry-standard PRD practices. Engineers (regular roles) use the ACP web app or IDE plugins to author UML/scripts and access the BRV dashboard in read-only mode for PRDs. The visual association editor in the BRV dashboard maps PRD requirements to UML elements for traceability. Role-based authorization (admin, manager, regular) restricts PRD edits to admin/manager roles in the web app, while all roles can view requirements. The BRV dashboard displays PRD requirements, UML diagrams, scripts, and validation results (Code Fidelity Score, RFS, optional security scans), ensuring >90% business-technical alignment, which is configurable. The Requirements Parser processes PRDs, and the Model Parser maps UML/scripts to a traceability matrix, visualized in the visual association editor. The CI/CD pipeline automates RFS validation, with manual review available via the dashboard for mismatches (e.g., missing integration logic). Data is stored in encrypted AWS S3.

The traceability matrix, generated by the Requirements Parser, is a JSON structure that maps PRD requirements to UML models and scripts, ensuring alignment between business needs and technical implementation. It includes the following fields:

- **req_id**: Unique identifier for each PRD requirement (e.g., "R1").
- **description**: Textual description of the requirement (e.g., "Process payments securely").
- **use_case**: Details of the use case, including embedded PlantUML for diagrams (e.g., sequence diagram for payment processing).
- **constraints**: Additional requirements like compliance (e.g., "GDPR-compliant").
- **uml_mappings**: Links to specific UML elements (e.g., PlantUML component/sequence diagram elements like "PaymentService -> AWS_S3: Log Transaction").
- **script_mappings**: Links to script functions (e.g., Python process_payment function).
- **rfs_score**: Requirements Fidelity Score (RFS), a percentage indicating alignment (target >90%).
- **status**: Validation status (e.g., "Validated," "Mismatch," "Pending").

An example traceability matrix entry is the following:

```
Example Requirements Traceability Matrix (JSON)
[
  {
    "req_id": "R1",
    "description": "Process payments securely",
    "use_case": {
      "description": "User initiates payment, system validates amount, logs
transaction to AWS S3, and publishes event to Kafka",
      "diagram": "@startuml\nactor User\nUser -> PaymentService: Initiate Payment\
nPaymentService -> AWS_S3: Log Transaction\nPaymentService -> Kafka: Publish Event\
n@enduml"
    },
    "constraints": [
      "GDPR-compliant",
      "No manual code edits"
    ],
    "uml_mappings": [
      {
        "element": "PaymentService -> AWS_S3: Log Transaction",
        "diagram_id": "seq_payment"
      },
      {
        "element": "PaymentService -> Kafka: Publish Event",
        "diagram_id": "seq_payment"
      }
    ],
    "script_mappings": [
      {
        "function": "process_payment",
        "file": "PaymentServiceLogic.py",
        "line": 10
      }
    ],
    "rfs_score": 92.5,
    "status": "Validated"
  }
]
```

The BRV Dashboard uses the traceability matrix to:

- **Visualize Alignment**: Displays mappings between PRD requirements, UML elements, and script functions via a visual association editor (e.g., linking "Process payments securely" to UML sequence messages).
- **Validate Requirements**: Integrates with the CI/CD pipeline’s RFS Validator to compute RFS (>90%) and flag mismatches (e.g., missing Kafka integration) for manual review.
- **Support Role-Based Interaction**: Allows Product Managers to edit PRDs (including requirements and use cases) and Engineers to view PRDs (read-only) and refine UML/scripts based on RFS feedback, with RBAC via OAuth 2.0 restricting PRD edits to admin/manager roles.

- **Facilitate Iteration**: Alerts users to low RFS or mismatches, enabling PRD edits or UML/script refinements, triggering targeted regeneration via the CI/CD pipeline.

The BRV Dashboard displays:

- **PRD Requirements**: List of requirements with req_id, description, use_case, and constraints (e.g., "R1: Process payments securely, GDPR-compliant").
- **UML Diagrams and Scripts**: Read-only views of PlantUML diagrams and script details (e.g., Python process_payment function) with mappings.
- **Validation Results**: RFS (>90%), Code Fidelity Score (CFS >90%), status (e.g., "Validated"), and optional security scan results (e.g., SonarQube outcomes).
- **Traceability Matrix**: Tabular/graphical view of mappings, highlighting mismatches.
- **Actionable Controls**: Buttons for Product Managers to edit PRDs, Engineers to refine UML/scripts, trigger “Show Mappings,” or “Generate Code.”
- **Environment Context**: Validation results per environment (Development, Staging, Production) and audit trails (via Kafka) for Production deployments.

4.6) Change Handling and Regeneration

To ensure model-driven purity and prevent regressions, ACP implements a robust change handling and regeneration process for UML models and scripts, enforcing no manual code edits:

- **Change Detection**: Model and script changes are detected using Git version control, leveraging git diffs to identify modified UML diagrams (e.g., PlantUML sequence messages) and scripts (e.g., Python functions). Changes are tracked in Git repositories (hosted in AWS CodeCommit or similar), with commits triggering CI/CD pipeline workflows (Sections 4.4, 4.9).
- **Targeted Regeneration**: The ACP Prompt-Based Build Engine regenerates only affected parts of the codebase, using ASTs from the Model Parser to identify modified elements (e.g., updated sequence messages or script functions). For large-scale systems, regeneration leverages the multi-prompt approach (Section 4.8), generating prompts only for affected subsystems to minimize overhead. Standardized prompt templates incorporate git diffs (e.g., updated UML/script AST nodes) to generate targeted code updates, ensuring non-destructive regeneration without regressions. The CI/CD pipeline validates regenerated code against Code Fidelity Score and RFS to confirm alignment.
- **Enforcement of No Manual Edits**: Manual code edits are prohibited via CI/CD pipeline gates, which reject code deviating from UML/script ASTs (e.g., via AST comparison in the ACP CFS

Validator). Deployments are read-only, ensuring immutability. Pre-commit hooks and optional SAST scans via an external Security Scanner (e.g., SonarQube) further enforce model-driven purity by rejecting non-compliant changes (Sections 3.1, 4.5). This process ensures traceability, prevents scope creep, and maintains >90% (configurable) business-technical alignment, integrated with the BRV dashboard for validation (Sections 4.5, 4.10).

4.7) Subsystem Support

To support flexible and risk-mitigated code generation, ACP implements subsystems as a functional requirement, supporting microservices, MCP-compliant servers, and agentic systems:

- Subsystem Modeling: Systems are modeled as UML subsystems (e.g., using PlantUML component diagrams) where modularity or complexity requires encapsulation (e.g., a service integrating external APIs). Subsystems are defined in UML component diagrams with <subsystem> stereotypes, ensuring clear boundaries for code generation and validation.
- Tools and Resources: Subsystems are generated as tools, prompts, or resources, configurable as one tool per service or shared across services. The ACP Prompt-Based Build Engine exposes UML/script mappings and PRD data as resources, orchestrated via REST/JSON prompts.
- Agentic and Traditional Flows: UML deployment diagrams include an option to select agentic or traditional flows for specific functionality areas, using stereotypes (e.g., <agentic> for LLM-driven dynamic invocation, <traditional> for standard execution). Agentic flows leverage LLM prompts to invoke subsystems dynamically (e.g., a service processing requests via prompts), while traditional flows use code deployed to AWS ECS without LLM reliance. This choice mitigates risks by allowing traditional flows for critical or stable functionality (e.g., core processing) and agentic flows for dynamic workflows (e.g., event-driven notifications), validated by CI/CD pipeline checks (Code Fidelity Score, RFS) to ensure >90% configurable alignment. This process enhances flexibility and mitigates risks through selective flow application, integrated with the BRV dashboard for business requirements validation (Sections 4.5, 4.10).

4.8) Incremental Code Generation

ACP's incremental code generation produces modular code for large-scale projects (e.g., 100,000+ lines) from UML (PlantUML) and scripts (Python, OCL) in version 1.0 of the ACP Model Parser, decomposing models into subsystems, generating code fragments via LLM prompts, and stitching them into a cohesive codebase. The process supports small-token LLMs (e.g., 4,096 tokens) through fine-grained decomposition and efficient stitching, ensuring scalability and model-driven purity.

- **Subsystem Decomposition**:
    - Model Parser splits UML models (e.g., component/sequence diagrams) into subsystems (e.g., microservices) using <subsystem> stereotypes, defining modular boundaries.
    - Generates Abstract Syntax Trees (ASTs) per subsystem, capturing UML elements (classes, relationships) and scripts.
    - For small-token LLMs, further subdivides subsystems into smaller units (e.g., classes, methods) to fit prompts within limits (e.g., 4,096 tokens).
    - Optionally processes visual UML images (LLaVA-1.5) into PlantUML text or ASTs via image parsing (e.g., custom diagram parser).
- **Multi-Prompt Generation**:
    - The Prompt-Based Build Engine MCP server creates JSON/YAML prompt templates per subsystem or sub-unit, targeting components (e.g., REST controllers, database logic).
    - Prompts embed compact ASTs and scripts, using minimal CoT/few-shot instructions to fit small token limits, ensuring consistent output (~10s/fragment).
    - Supports text-based PlantUML (default) and visual UML inputs (LLaVA-1.5), with tailored prompts (e.g., {"prompt": "Given UML AST: {ast_data}, generate Java/Spring endpoint..."}).
    - Managed via LiteLLM for LLM-agnosticism, enabling rapid model switching (Section 4.13).
- **Prompt Orchestration**:
    - Orders subsystem/sub-unit generation based on UML AST dependencies (e.g., shared models before services).
    - Executes prompts sequentially (small projects) or in parallel (large projects, via AWS Lambda), caching dependencies in memory (Development) or ElastiCache (Staging/Production) to fit token limits (Section 4.9).
    - Runs locally in Docker (Development, HTTP/REST) or on AWS ECS with gRPC (Staging/Production) for high throughput.
- **Code Fragment Stitching**:
    - Merges fragments into a cohesive codebase (e.g., Spring Boot, Flask project) using a predefined project structure (e.g., Maven/Gradle directories).

    - Inserts boilerplate (e.g., pom.xml, application.properties) via templating (e.g., Jinja2) for consistency.
    - Resolves dependencies and imports using AST analysis, deduplicating shared code (e.g., common classes) with indexed lookups to handle high fragment counts (e.g., 100+ for 100,000 lines).
    - In Development, stitches locally, caching results for “Show Mappings”/“Generate Code” buttons (~1-2s overhead for 100 fragments); in Staging/Production, parallelizes via AWS Lambda.
    - Outputs a deployable codebase (e.g., zipped project or CodeCommit push) for validation (section 4.9).
- **Incremental Updates**:
    - Git diffs in AWS CodeCommit identify modified UML/script AST nodes, triggering regeneration for affected subsystems/sub-units only.
    - Regenerates fragments (~10s/fragment), re-stitches into the existing codebase, preserving unchanged components via AST-based merging.
    - Supports small-token LLMs by limiting prompts to modified units, maintaining efficiency.

4.9) Deployment Workflow for Multiple Environments

The ACP platform supports Development, Staging, and Production environments, ensuring model-driven purity, traceability, and secure access via MCP (HTTP/REST over HTTPS, WebSocket, gRPC), OAuth 2.0, and role-based access control (RBAC). Only engineers (Regular/Admin) add/modify UML/scripts in the IDE plugin/web app’s graphical editor and access read-only PRD data in BRV. Product Managers (Managers) view UML read-only and create/edit PRDs in BRV via the IDE plugin/web app.

1. **Development Environment**:
    - **Build Engine**: MCP server in a Docker container on the engineer’s local PC, using HTTP/REST over HTTPS to communicate with a cloud-hosted LLM (e.g., in AWS ECS). Lightweight CFS/RFS validators run locally, caching results for on-demand UML/code cross-referencing via a “Show Mappings” button (real-time optional) and code generation via a “Generate Code” button in the ACP IDE plugin and web app.

- **Workflow**: Engineers (Regular/Admin) author UML (PlantUML) and scripts (Python, OCL) via the ACP IDE plugin/web app, with read-only PRD access in BRV. "Show Mappings" displays UML/code mappings; "Generate Code" triggers generation and local CFS/RFS validation (>90%, configurable). Code is cached locally, committed to AWS CodeCommit in batches. Managers view UML read-only and edit PRDs in BRV via the ACP web app, without code generation access. OAuth 2.0 (AWS Cognito) and RBAC enforce access.
- **Rationale**: Local Build Engine and validators enable responsive on-demand workflows; CodeCommit ensures traceability.

2. **Staging Environment**:
    - **Build Engine**: MCP server in a Docker container on AWS ECS, using HTTP/REST, WebSocket, and gRPC for parallel generation via AWS Lambda.
    - **Workflow**: Engineers (Regular/Admin) author UML/scripts; Managers view UML read-only and edit PRDs in BRV. Only engineers trigger code generation; validated code (CFS, RFS >90% configurable, SonarQube) deploys to AWS ECS. OAuth 2.0 and mTLS secure MCP endpoints. Configurations use AWS Secrets Manager.
    - **Rationale**: gRPC enhances performance; cloud validators reduce local overhead.
3. **Production Environment**:
    - **Build Engine**: MCP server in a Docker container on a dedicated AWS ECS cluster, prioritizing gRPC (HTTP/REST, WebSocket as fallbacks).
    - **Workflow**: Engineers (Regular/Admin) author UML/scripts; Managers view UML read-only and edit PRDs in BRV. Only engineers (Admin only for deployments) trigger generation; validated code (CFS, RFS >90% configurable, SonarQube) deploys to AWS ECS read-only, stored in encrypted S3, with Kafka for audit trails. OAuth 2.0, mTLS, and RBAC secure access. Configurations are in AWS Secrets Manager.
    - **Rationale**: gRPC and cloud infrastructure ensure scalability and security.
4. **Cross-Environment Consistency**:
    - CI/CD enforces CFS/RFS validation. UML/Requirements Parsers maintain consistent ASTs and traceability. LLM-agnostic prompts ensure uniform code generation. BRV validates PRDs, with MCP securing communication.
5. **Performance and Security**:

- Code generation (~10s/fragment) and validation (~1min/component) meet targets. MCP uses OAuth 2.0 (AWS Cognito), mTLS (Staging/Production), and TLS 1.3 for transport security. RBAC restricts UML/generation to engineers, PRD edits to Managers, Production deployments to Admins. Encrypted storage (S3, Secrets Manager) and OWASP-compliant SAST scans increase security compliance.

4.10) IDE Plugins

ACP IDE plugins (VS Code, IntelliJ, with PlantUML extensions) enable engineers to author UML/scripts, view the BRV dashboard, and refine mappings. Plugins integrate with the Model Parser and ACP Prompt-Based Build Engine via APIs, supporting local and cloud workflows. The BRV dashboard mirrors the ACP web app's functionality, allowing engineers to review PRD-to-UML alignment and validation results (Code Fidelity Score, RFS, optional security scans) in read-only mode for PRD requirements, with PRD editing disabled to enforce Product Manager web app exclusivity.

4.11) Error Handling and Recovery

- **Parsing Errors**: The Model Parser logs model syntax errors, with suggestions provided via the ACP web app or IDE plugin. Engineers correct UML/scripts based on detailed logs.
- **Validation Errors**: The CI/CD pipeline rejects code with low Code Fidelity Score or RFS, logging discrepancies for refinement. Alerts are sent to the BRV dashboard (web app/IDE).
- **Recovery**: Rollback to previous UML/script versions via Git ensures recovery from failed deployments. The ACP Prompt-Based Build Engine regenerates code from corrected models.
- **Debugging**: The ACP IDE plugin maps code runtime errors to UML elements/scripts, enabling the location of model-level fixes.

4.12) Security

The platform ensures security compliance by design:

- **Model-Driven Purity**: Eliminates manual code edits, reducing vulnerabilities (e.g., SQL injection). Read-only deploys eliminate manual edits, enforced by CI/CD pipeline gates (Section 4.4).
- **CI/CD Pipeline Security**: Integrates optional external SAST Security Scanner (OWASP-compliant, e.g., SonarQube) and required RFS checks to ensure PRD-aligned, secure code.
- **Secure Storage**: PRDs, UML, and scripts are stored in encrypted AWS S3 with RBAC via AWS IAM.

- **Authentication**: Role-based authorization (admin, manager, regular) restricts PRD edits to admin/manager roles in the ACP web app.

4.13) LLM Switching Process

The LLM switching process leverages LiteLLM, a lightweight abstraction layer, to standardize API calls across open-source and proprietary models, such as DeepCode, GPT-4, or others. The process ensures compatibility with ACP's JSON/YAML prompt templates, which incorporate Chain-of-Thought (CoT) and few-shot learning for consistent code generation. The steps are as follows:

1. **Configuration Update**: Update the LiteLLM configuration to point to the new LLM's API endpoint or local deployment (e.g., DeepCode hosted on AWS ECS). This involves modifying the model identifier and authentication credentials, typically completed in ~5 minutes.
2. **Prompt Revalidation**: Revalidate standardized prompt templates against the new LLM to ensure compatibility with UML-to-code generation tasks. ACP's default LLM, DeepCode, is optimized for code generation, processes text-based UML representations (e.g., PlantUML source code or ASTs generated by the Model Parser). Revalidation adjusts prompts for DeepCode's tokenization and response style.
3. **LoRA Fine-Tuning (Optional)**: For domain-specific customization, apply Low-Rank Adaptation (LoRA) to fine-tune the new LLM using a dataset of ~1,000 UML-to-code examples (e.g., PlantUML diagrams/scripts, ASTs to Java/Spring code). DeepCode's open-source weights enable efficient LoRA fine-tuning on cloud infrastructure, completing in 20–30 minutes for a 33B-parameter model. Fine-tuning ensures a Code Fidelity Score (CFS) >90% (configurable) by aligning DeepCode with ACP's UML-driven workflow (Sections 4.3, 4.4).
4. **Validation**: Test the new LLM's outputs using the CI/CD Pipeline, which validates code fragments against UML/script ASTs (CFS) and PRD requirements (RFS) via the LLM. For DeepCode, validation confirms that generated code (e.g., Python/Flask endpoints) matches preprocessed UML ASTs, ensuring model-driven purity. This step takes ~10–15 minutes, including unit/integration tests with mock dependencies.

The best-case scenario of total switching time, including fine-tuning, is approximately one hour, meeting ACP's performance requirements (Section 2.3). If fine-tuning is skipped (e.g., for DeepCode's strong baseline code generation), switching can complete in ~20–30 minutes.

4.13.1) DeepCode as the Default LLM

DeepCode, an open-source, code-focused LLM, is the default choice for ACP due to its superior out-of-the-box code generation capabilities, permissive licensing, and compatibility with LoRA fine-tuning.

Unlike multimodal models like LLaVA-1.5, DeepCode is text-based, requiring the Model Parser to preprocess visual UML diagrams (e.g., PlantUML renderings) into text-based ASTs. This preprocessing ensures accurate mapping of UML models to ASTs, helping to achieve CFS >90% (configurable). DeepCode's moderate size (e.g., 33B parameters) supports efficient inference on AWS Lambda/ECS and cost-effective fine-tuning, aligning with economic feasibility (Section 3.3).

4.13.2) Alternate LLMs

ACP supports switching (Section 4.13) to alternative LLMs, such as LLaVA-1.5 (for multimodal UML processing), GPT-4 (via API), or Mixtral (open-source). Each requires revalidation of prompts and, for open-source models, optional LoRA fine-tuning. For example, switching to LLaVA-1.5 leverages its ability to process visual UML diagrams directly, while GPT-4 relies on prompt engineering due to its proprietary nature. The CI/CD Pipeline ensures that all LLMs produce code meeting CFS and RFS thresholds.

4.13.3) Considerations

- **Preprocessing for LLM**: The Model Parser converts visual UML diagrams and supporting scripts into text-based representations as AST files, adding ~5 seconds per diagram but maintaining accuracy. Engineers can refine UML inputs via the ACP web app or IDE plugins to address parsing errors.
- **Fine-Tuning Dataset**: A high-quality dataset of UML-to-code pairs, stored in the Git repository, is critical for LoRA fine-tuning. DeepCode's code-focused training reduces dataset size requirements compared to general-purpose models.
- **Performance**: DeepCode's inference speed supports ~10-second code generation per fragment, with parallel execution via AWS Lambda for large systems (Sections 4.8, 4.9).
- **Enterprise Control**: DeepCode's open-source nature allows deployment on secure AWS ECS clusters, meeting RBAC and security requirements (Section 4.12).

4.13.4) Future-Proofing

The LLM-agnostic design ensures ACP can adopt emerging models (e.g., future DeepSeek or xAI models like Grok 4) by repeating the switching process. The Business Requirements Validation (BRV) Dashboard tracks LLM performance (CFS, RFS) to guide future switches.

4.14) Sample UML and Scripting for Code Generation

This payment processing example illustrates how ACP translates PRD and UML into code, applicable to any system use case. Below is a sample UML sequence diagram (PlantUML), its supporting Python script, and PRD requirement for a payment processing microservice integrating AWS S3 (for logs) and Kafka (for events), showing how BRV automatically validates alignment with the product requirements.

Usage in Agile7's System:

- Product Manager inputs the PRD requirement via the ACP web app's BRV section; the engineer submits the PlantUML diagram and its supporting Python script via the ACP web app or IDE plugin.
- The ACP Prompt-Based Build Engine generates a Spring Boot microservice, validated against the PRD via RFS in the CI/CD pipeline.
- The BRV dashboard (web app or IDE plugins) shows alignment (e.g., S3 logging, Kafka publishing) or flags mismatches for refinement.

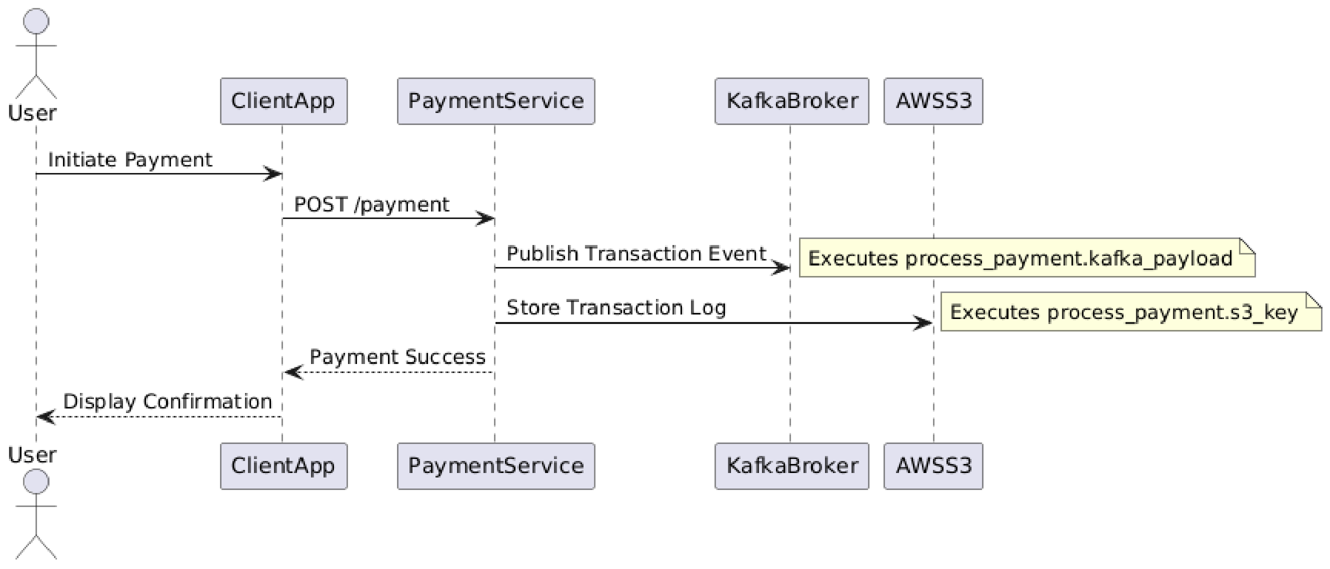
PRD Requirement (JSON):

```
{
  "req_id": "R1",
  "description": "Process payments securely",
  "use_case": {
    "description": "User initiates payment, system validates amount, logs 
transaction to AWS S3, and publishes event to Kafka",
    "diagram": "@startuml\nactor User\nUser -> PaymentService: Initiate Payment\
nPaymentService -> AWS_S3: Log Transaction\nPaymentService -> Kafka: Publish Event\
n@enduml"
  },
  "priority": "High",
  "constraints": ["GDPR-compliant", "No manual code edits"]
}
```

UML Sequence Diagram (PlantUML):

The UML Sequence Diagram for this example is shown in Figure 3 below. Its corresponding PlantUML code is in the Appendix, Section A3 and can be viewed by copying and pasting it into the form at https://www.plantuml.com/plantuml/.

UML Diagram for Code-Generation Example (Figure 3)



Supplemental Script (Python for Logic):

```
# PaymentServiceLogic.py
# Defines payment processing logic for the PaymentService microservice

from datetime import datetime
import uuid

def process_payment(amount, user_id):
    # Validate payment amount
    if amount <= 0:
        raise ValueError("Invalid payment amount")

    # Generate transaction ID
    transaction_id = str(uuid.uuid4())

    # Kafka event payload
    event = {
        "transaction_id": transaction_id,
        "user_id": user_id,
        "amount": amount,
        "timestamp": datetime.now().isoformat()
    }
    # AWS S3 storage details
    s3_key = f"transactions/{transaction_id}.json"
    s3_bucket = "payment-logs"

    return {
        "kafka_topic": "payment-events",
        "kafka_payload": event,
        "s3_bucket": s3_bucket,
        "s3_key": s3_key
    }
```

BRV Validation:

- The Requirements Parser extracts "Process payments securely," "validate amount," "log to AWS S3," and "publish to Kafka" from the PRD.
- The traceability matrix maps these to UML (e.g., Payment -> S3: Store Transaction Log, Payment -> Kafka: Publish Transaction Event) and script (e.g., process_payment with validation).
- The CI/CD pipeline computes an RFS (>90% indicates alignment); mismatches (e.g., missing validation) are flagged in the BRV dashboard for manual review via the ACP web app or IDE plugins.

Generated Java Code:

```
package com.agile7.payment;

import com.fasterxml.jackson.databind.ObjectMapper;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import java.util.UUID;
import java.time.Instant;
import java.util.Map;

@RestController
public class PaymentController {
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;
    @Autowired
    private S3Client s3Client;
    private final ObjectMapper mapper = new ObjectMapper();

    @PostMapping("/payment")
    public String processPayment(@RequestBody Map<String, Object> payload) {
        double amount = ((Number) payload.get("amount")).doubleValue();
        String userId = (String) payload.get("userId");

        if (amount <= 0) {
            throw new IllegalArgumentException("Invalid payment amount");
        }

        String transactionId = UUID.randomUUID().toString();
        Map<String, Object> event = Map.of(
            "transactionId": transactionId,
            "userId": userId,
            "amount": amount,
            "timestamp": Instant.now().toString()
        );

        // Publish to Kafka
        try {
            kafkaTemplate.send("payment-events", mapper.writeValueAsString(event));
        } catch (Exception e) {
            throw new RuntimeException("Failed to publish to Kafka: " +
e.getMessage());
        }

        // Store in S3
        String s3Key = "transactions/" + transactionId + ".json";
        try {
            s3Client.putObject(PutObjectRequest.builder()
                .bucket("payment-logs")
                .key(s3Key)
                .build(), mapper.writeValueAsString(event));
        } catch (Exception e) {
            throw new RuntimeException("Failed to store in S3: " + e.getMessage());
        }

        return "Payment Success: " + transactionId;
    }
}
```

Maven Dependencies (pom.xml):

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <version>3.2.5</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
        <version>3.1.4</version>
    </dependency>
    <dependency>
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>s3</artifactId>
        <version>2.25.16</version>
    </dependency>
    <dependency>
```

```
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.16.1</version>
    </dependency>
</dependencies>
```

5) Model Parser Release Roadmap

This section outlines the progressive software releases for the Model Parser, a core architectural component integrated into the Agile7 Code Platform (ACP) web app and IDE plugin. The Model Parser interprets modeling artifacts (e.g., UML diagrams, scripts, and later DSLs (Domain-Specific Language)) to generate Abstract Syntax Trees (ASTs) for the ACP Prompt-Based Build Engine, enabling LLM-driven code generation across multiple languages/frameworks. Releases build incrementally, starting with Version 1.0's focus on importing UML (PlantUML) diagrams from third-party graphical editors and editing UML text and Python scripts within the ACP web app/IDE plugins, ensuring model-driven purity and business-technical alignment.

File-format backward compatibility is maintained using extensible textual formats (e.g., .puml for PlantUML, .py for scripts, evolving to .mdl or .dsl with JSON/YAML wrappers). Older files load seamlessly in newer versions via automated migration tools, preserving data integrity and enabling architects/developers to upgrade without rework. Version 1.0 supports importing .puml files from third-party graphical UML editors (e.g., Visual Paradigm, Lucidchart) with automated syntax/error checking.

The final release targets enterprise systems, incorporating scalability, security, and advanced integrations. User-friendly interfaces (e.g., text editors, validation tools, and later graphical editors) ensure architects and developers can create LLM-ready models efficiently, aligning with the ACP whitepaper's goals.

Version 1.0: Initial Release (Alpha Stage – Foundational Modeling Support)

- **Core Features:**
    - Parses PlantUML diagrams (.puml files) imported from third-party graphical UML editors (e.g., Visual Paradigm, Lucidchart) and supporting Python scripts (.py) to generate ASTs, supporting basic subsystem decomposition using UML stereotypes (e.g., `<subsystem>`).
    - Visual medium: No graphical UML editor in the ACP web app/IDE plugins; instead, supports importing .puml files with automated syntax and error checking. Includes a

    text-based PlantUML editor, and a Python script editor with template snippets for logic (e.g., payment processing). Provides a preview pane for AST tree structures.

  - Supports code generation for 2–3 languages (e.g., Java/Spring, Python/Flask) via the Prompt-Based Build Engine.
  - Handles third-party integrations (e.g., AWS S3, Kafka) via UML-defined interactions, validated for syntax/errors (<2% error rate, Section 3.1).
  - Uses DeepCode LLM with LoRa fine-tuning for generating ASTs.

- **User-Friendliness:** Text-based editors for PlantUML and Python with real-time syntax validation and error suggestions reduce ambiguity. Engineers import .puml files from familiar third-party tools or edit text-based UML and scripts directly in the ACP environment.
- **Backward Compatibility:** Baseline .puml and .py file formats, exportable as XML/JSON ASTs (.mdl) for interoperability with PlantUML servers and third-party tools.
- **LLM Integration:** Generates structured ASTs for standardized prompts, supporting basic traceability to PRD requirements via the Requirements Parser (Section 4.5). These ASTs will have a medium level of ambiguity in this first release, suitable for small projects.
- **Release Focus:** Proof-of-concept for UML-to-AST parsing and code generation, tested in small-scale environments with ~10-second processing per diagram/script.

Version 1.1: Enhanced Release (Beta Stage – Reduced Ambiguity and Expanded Inputs)

- **Core Features:**
  - Extends parsing to include diagrams-as-code (e.g., Mermaid) alongside PlantUML, with initial DSL templates for prompt engineering (e.g., JSON-based constraints).
  - Visual medium: Adds a graphical editor in the ACP web app/IDE plugins for Mermaid diagrams with drag-and-drop support, while continuing to import .puml files with enhanced error checking (e.g., semantic validation for subsystem boundaries). Text-based editors for PlantUML, Mermaid, and DSL templates include AI-assisted suggestion pop-ups for ambiguous elements and a DSL template gallery with clickable fields. Real-time collaboration overlay supports team-based editing.
  - Supports AST generation for 5+ languages/frameworks, with automated error handling and edge-case validation.

    - AI-assisted refinement minimizes LLM hallucinations (~5% risk post-RAG, Section 3.1).
    - Uses DeepCode LLM with LoRa fine-tuning for generating ASTs.
- **User-Friendliness:** Graphical Mermaid editor and guided DSL wizards ease adoption, while .puml imports maintain familiarity. Collaboration tools and error suggestions enhance usability.
- **Backward Compatibility:** Supports v1.0 .puml/.py files; new .mdlx format for Mermaid-inclusive models degrades gracefully in older versions.
- **LLM Integration:** Enhances ASTs with metadata for task-specific prompts, including code review hints.
- **Release Focus:** Beta testing emphasizes parsing reliability and integration with CI/CD and BRV for >90% configurable Requirements Fidelity Score (RFS, Section 4.5).

Version 2.0: Mature Release (Release Candidate Stage – Comprehensive Parsing Support)

- **Core Features:**
    - Comprehensive DSL editor with formal grammars, parsing multimodal inputs (e.g., natural language to UML/DSL via NLP).
    - Graphical DSL editor with drag-and-drop widgets for grammar definition (flowcharts/trees), alongside Mermaid/PlantUML editor for hybrid inputs. Multimodal canvas supports text/visual/voice inputs, and testing stub visualizer.
    - Generates ASTs for 10+ languages/frameworks, including secure coding practices and automated testing stubs.
    - Supports incremental parsing for large-scale systems, decomposing models into subsystems (Section 4.8).
    - Uses DeepCode LLM with LoRa fine-tuning as default for generating ASTs. Supports other LLMs, integrated through semi-automated training and prompt adjustment, and LoRa. (Section 4.13)
- **User-Friendliness:** No-code interfaces for non-experts, AI-optimized suggestions, and tutorials. UML users continue to be supported via familiar .puml imports.

- **Backward Compatibility:** Imports v1.0 and v1.1 files (.puml, .py, .mdlx) to .dsl formats via parser transformation rules.
- **LLM Integration:** Advanced AST structuring for fine-tuned prompts, including enterprise pre-training data.
- **Release Focus:** Pre-enterprise validation, focusing on performance in multi-user scenarios and Prompt-Based Build Engine integration (Section 4.10).

Version 3.0: Enterprise Release (General Availability – Scalable and Secure Parsing)

- **Core Features:**
    - Full integration with enterprise tools (CI/CD, AWS), role-based access control, and compliance features (e.g., GDPR via encrypted parsing).
    - Enterprise-grade UI with graphical canvas for DSLs, Mermaid, and PlantUML; supports zoom/pan, 3D visualization, and multimodal visual processing (e.g., LLaVA-1.5 for UML images). Analytics dashboard with interactive charts for parsing performance and CI/CD visualization.
    - Supports >90% RFS for mission-critical systems (Section 4.5).
    - Uses DeepCode LLM with LoRa fine-tuning as default for generating ASTs. Supports other LLMs, integrated through full-automated training and prompt adjustment, and LoRa. (Section 4.13)
- **User-Friendliness:** Customizable UI, APIs for integrations, and training modules for architects. Continued support for UML imports.
- **Backward Compatibility:** Comprehensive support for all prior formats (.puml, .py, .mdlx, .dsl) with automated migration tools.
- **LLM Integration:** Multimodal parsing, code review automation, and fine-tuning for high-fidelity ASTs.

Glossary

- ACP: Agile7 Code Platform
- API: Application Programming Interface, a set of rules for software components to communicate.
- AST: Abstract Syntax Tree, a tree representation of the syntactic structure of code. ACP produces language-agnostic ASTs.
- AWS: Amazon Web Services, a cloud platform for hosting and storage.
- BRV: Business Requirements Validation, ensuring PRD-UML alignment.
- CI/CD: Continuous Integration/Continuous Deployment, automated processes for code integration and deployment.
- CodeGen: Code Generation, encompassing initial creation and regeneration of code.
- CFS: Code Fidelity Score
- DSL: Domain-Specific Language, a programming language designed to solve problems within a specific domain
- GDPR: General Data Protection Regulation, a regulation for data protection and privacy.
- IAM: Identity and Access Management, a framework for controlling access to resources.
- LLM: Large Language Model, an AI model for processing and generating text or code.
- LLM-Agnostic: Independent of specific language models, using abstractions like LiteLLM.
- LoRA: Low-Rank Adaptation, a technique for fine-tuning large language models efficiently.
- MCP: Model Context Protocol, an open protocol for connecting LLMs to external data sources, tools, and services in a unified way.
- MDE: Model-Driven Engineering, a methodology using models as the primary development artifact.
- NLP: Natural Language Processing, AI techniques for understanding and processing human language.
- OCL: Object Constraint Language, used for defining UML constraints.
- PCI-DSS: Payment Card Industry Data Security Standard, a standard for securing payment data.

- PRD: Product Requirements Document, a document specifying business requirements and use cases.
- RAG: Retrieval-Augmented Generation, enhancing LLM prompts with external data.
- RBAC: Role-Based Access Control, a method for regulating access based on user roles.
- RFS: Requirements Fidelity Score, measuring PRD-UML semantic match.
- SAST: Static Application Security Testing, a method for analyzing code for vulnerabilities.
- UML: Unified Modeling Language, a standard for visualizing system designs.

References

1 OWASP. (2025). *OWASP Dependency-Check Documentation*. Open Web Application Security Project. Retrieved from https://owasp.org/www-project-dependency-check

2 SonarSource. (2025). *SonarQube User Guide*. SonarSource. Retrieved from https://www.sonarqube.org

3 European Union. (2025). *General Data Protection Regulation (GDPR)*. Retrieved from https://gdpr.eu

4 PCI Security Standards Council. (2025). *Payment Card Industry Data Security Standard (PCI-DSS)*. Retrieved from https://www.pcisecuritystandards.org

5 BerriAI. (2025). *LiteLLM: Python SDK, Proxy Server (LLM Gateway)*. Retrieved from https://litellm.ai

6 Amazon Web Services. (2025). *Amazon S3 User Guide*. Amazon Web Services. Retrieved from https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

7 Amazon Web Services. (2025). *Amazon Elastic Container Service Developer Guide*. Amazon Web Services. Retrieved from https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

8 Amazon Web Services. (2025). *AWS Identity and Access Management (IAM) User Guide*. Amazon Web Services. Retrieved from https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

9 Amazon Web Services. (2025). *Amazon SageMaker Developer Guide*. Amazon Web Services. Retrieved from https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html

10 DeepSeek. (2025). *DeepSeek-Coder Documentation*. DeepSeek. Retrieved from https://github.com/deepseek-ai/DeepSeek-Coder

11 Apache Software Foundation. (2025). *Apache Kafka Documentation*. Retrieved from https://kafka.apache.org/documentation

12 Git SCM. (2025). *Git Reference Manual*. Retrieved from https://git-scm.com/docs

13 CloudBees. (2025). *Jenkins User Handbook*. Retrieved from https://www.jenkins.io/doc

14 Object Management Group. (2025). *OMG Object Constraint Language (OCL), Version 2.4*. Retrieved from https://www.omg.org/spec/OCL/2.4

15 Microsoft. (2025). *Visual Studio Code Extension API*. Retrieved from https://code.visualstudio.com/api

16 JetBrains. (2025). *IntelliJ IDEA User Guide*. Retrieved from https://www.jetbrains.com/help/idea

17 PlantUML Team. (2025). *PlantUML Language Reference Guide*. PlantUML. Retrieved from http://plantuml.com

Appendix

A.1) ACP Architecture Diagram's PlantUML Code

```
@startuml
title ACP Architecture Diagram (Figure 1)
!define RECTANGLE class
skinparam linetype ortho
skinparam ranksep 100

package "ACP Web/IDE App" {
  [ACP Web Portal] -down-> [BRV Dashboard]
  [ACP Web Portal] -down-> [Model Parser]
  [ACP Web Portal] -down-> [Requirements Parser]
  [ACP IDE Plugin] -down-> [BRV Dashboard]
  [ACP IDE Plugin] -down-> [Model Parser]
  [ACP IDE Plugin] -down-> [Requirements Parser]
  [BRV Dashboard] -down-> [Requirements Parser]
  [BRV Dashboard] -down-> [Model Parser]

  note right of [BRV Dashboard]
    Visualizes PRD-UML mappings
    Displays CFS, RFS, and validation
  end note

  note right of [Requirements Parser]
    Processes PRD (JSON, YAML, NL)
    Creates traceability matrix
  end note
}

package "ACP Core System Components" {
  [Model Parser] -down-> [Prompt-Based Build Engine]
  [Requirements Parser] -down-> [Prompt-Based Build Engine]
  [Prompt-Based Build Engine] -down-> [CI/CD Pipeline]
  [Prompt-Based Build Engine] -down-> [RAG Module]
  [CI/CD Pipeline] -down-> [Security Scanner]
  [RAG Module] -down-> [External APIs]

  note right of [Prompt-Based Build Engine]
    MCP Server with LLM
    JSON/YAML Prompt Templates
    LoRA Fine-Tuning
  end note

  note right of [CI/CD Pipeline]
    Validates CFS (>90%) and RFS (>90%)
    Integrates SAST
  end note
  note right of [Model Parser]
    Processes PlantUML and scripts
    Generates ASTs
  end note

}

package "External Systems" {
  [AWS S3] -down-> [Requirements Parser]
  [AWS S3] -down-> [Prompt-Based Build Engine]
  [AWS CodeCommit] -down-> [Model Parser]
  [AWS CodeCommit] -down-> [CI/CD Pipeline]
  [AWS ECS] -down-> [Deployed Code]
  [External APIs]
}

' Actors at top
actor "Product Manager" as PM
actor "Engineer" as ENG

PM -down-> [ACP Web Portal]
ENG -down-> [ACP Web Portal]
ENG -down-> [ACP IDE Plugin]

@enduml
```

A.2) ACP Sequence Diagram's PlantUML Code

```
@startuml
title ACP Sequence Diagram (Figure 2)
skinparam sequenceArrowThickness 2
skinparam roundcorner 10
skinparam maxmessagesize 150
skinparam sequenceParticipant underline

actor ProductManager
actor Engineer
participant WebApp as "ACP Web Portal / IDE Plugin"
participant ReqParser as "Requirements Parser"
participant UMLParser as "Model Parser"
participant BuildEngine as "Prompt-Based Build Engine"
participant RAG as "RAG Module"
participant CICDPipeline as "CI/CD Pipeline"
participant Security as "Security Scanner (Optional)"
participant BRVDashboard as "BRV Dashboard"

' Cloud/External systems
participant AWS_CodeCommit
participant AWS_Lambda
participant AWS_S3
participant Kafka
participant AWS_ECS

' Input Processing (Section 4.4, Step 1)
ProductManager -> WebApp: Enter/Edit PRD (JSON/YAML/NL/PlantUML) via forms/drag-
and-drop
WebApp -> ReqParser: Process PRD into traceability matrix
activate ReqParser
ReqParser --> WebApp: Return traceability matrix
deactivate ReqParser
WebApp -> AWS_CodeCommit: Store PRD data
WebApp -> BRVDashboard: Update with PRD and traceability matrix

Engineer -> WebApp: View PRD (read-only via BRV)
Engineer -> WebApp: Author/Update UML (PlantUML) and scripts (Python/OCL)
WebApp -> UMLParser: Parse UML/scripts into ASTs
activate UMLParser
UMLParser --> WebApp: Return ASTs (decomposed into subsystems)
deactivate UMLParser
WebApp -> AWS_CodeCommit: Commit UML/scripts

' Code Generation (Section 4.4, Step 2)
Engineer -> WebApp: Trigger "Generate Code" with ASTs and traceability matrix
WebApp -> BuildEngine: Request code generation (UML/script ASTs, traceability
matrix)
activate BuildEngine
BuildEngine -> RAG: Retrieve external data (e.g., AWS S3/Kafka API docs)
activate RAG
RAG --> BuildEngine: Return context for prompts
deactivate RAG
BuildEngine -> BuildEngine: Generate code fragments (multi-prompt, LLM/LoRA)
BuildEngine -> AWS_Lambda: Parallel generation for subsystems
BuildEngine --> WebApp: Return stitched codebase
deactivate BuildEngine

' Validation (Section 4.4, Step 3)
WebApp -> CICDPipeline: Submit codebase for validation
activate CICDPipeline
CICDPipeline -> Security: Run optional SAST scan (SonarQube, OWASP-compliant)
activate Security
Security --> CICDPipeline: Return scan results
deactivate Security
CICDPipeline -> CICDPipeline: Validate Code Fidelity Score (CFS >90% via AST
comparison)
CICDPipeline -> CICDPipeline: Validate Requirements Fidelity Score (RFS >90% via
traceability matrix)
CICDPipeline --> WebApp: Return validation results (CFS, RFS, scans)
deactivate CICDPipeline

' Stitching and Storage (Section 4.4, Step 4)
```

```
WebApp -> BuildEngine: Store stitched codebase
activate BuildEngine
BuildEngine -> AWS_S3: Store codebase (encrypted)
BuildEngine -> Kafka: Stream metadata
deactivate BuildEngine

' Deployment (Section 4.4, Step 5)
CICDPipeline -> AWS_ECS: Deploy validated codebase
CICDPipeline -> BRVDashboard: Update with CFS, RFS, and PRD alignment

' Review and Refinement
ProductManager -> BRVDashboard: Review RFS and PRD alignment, edit PRD if needed
Engineer -> BRVDashboard: Review CFS/RFS, refine UML/scripts if mismatches

' Regeneration (Section 4.4, Step 6)
Engineer -> WebApp: Commit UML/script updates
WebApp -> AWS_CodeCommit: Detect changes via git diffs
AWS_CodeCommit -> CICDPipeline: Trigger targeted regeneration
CICDPipeline -> BuildEngine: Regenerate affected subsystems
activate BuildEngine
BuildEngine --> CICDPipeline: Return updated codebase
deactivate BuildEngine
CICDPipeline -> AWS_ECS: Deploy updated codebase
CICDPipeline -> BRVDashboard: Update with new validation results

note over WebApp, CICDPipeline
Workflow ensures model-driven purity, no manual edits,
and >90% PRD alignment (Sections 4.4, 4.6)
end note
@enduml
```

A.3) ACP Code-Generation Example UML Diagram

```
@startuml PaymentProcessing
title UML Diagram for Code-Generation Example (Figure 3)
actor User
participant "ClientApp" as Client
participant "PaymentService" as Payment
participant "KafkaBroker" as Kafka
participant "AWSS3" as S3

User -> Client: Initiate Payment
Client -> Payment: POST /payment
Payment -> Kafka: Publish Transaction Event
note right: Executes process_payment.kafka_payload
Payment -> S3: Store Transaction Log
note right: Executes process_payment.s3_key
Payment --> Client: Payment Success
Client --> User: Display Confirmation
@enduml
```

Please contact Agile7 at **info@agile7.com**.

About Agile7

Agile7, founded by Max Meinhardt, delivers innovative software architecture and development services, specializing in energy-efficient, secure systems. Learn more at **www.agile7.com**.

Document Revision History

Author	Version	Date	Description
Max Meinhardt	1.0b	Sep 2, 2025	Initial beta release

Author Biography

About Max Meinhardt

Max Meinhardt is the founder of Agile7, his professional brand specializing in software engineering. With over three decades of industry experience, Max has led the architecture and implementation of enterprise web applications and carrier-class networking and telecommunications equipment firmware.

Max holds a BS in Computer Engineering Technology from Rochester Institute of Technology and an MBA from Thunderbird School of Global Management.

Max's LinkedIn Account: https://www.linkedin.com/in/maxmeinhardt/