CLG Technical Specification

Executive Summary

CLG (Composable Landing Generator) is a multi-brand landing page system built for Access Group. It enables marketing teams to create and manage landing pages across multiple brands using a visual editor, while maintaining consistent brand theming, equipment enquiry functionality, and visitor tracking capabilities.

The system combines a Next.js frontend with Builder.io as the headless CMS, Google Cloud Platform for serverless visitor identification, and Vercel for hosting with continuous deployment.

System Architecture

Overview

The architecture follows a composable approach where content management, visitor tracking, and frontend rendering are handled by separate specialised services that communicate via APIs. This separation allows each component to scale independently and be updated without affecting others.

Frontend Application

The frontend is built on Next.js using the App Router architecture. All pages are server-side rendered to ensure optimal search engine indexing and fast initial page loads. The application uses dynamic routing with catch-all segments to handle any URL structure defined in the CMS.

When a request comes in, the server fetches the corresponding page content from Builder.io, resolves the brand theme, and renders the page with the appropriate styling. The rendered HTML includes the brand's CSS variables, making theming instantaneous without client-side JavaScript overhead.

Content Management System

Builder.io serves as the headless CMS, providing both content storage and a visual editing interface. Content editors can drag and drop registered components onto pages, configure their properties, and preview changes in real-time. The system uses two primary data models: one for pages and one for brands.

Pages store the visual layout, component configurations, SEO metadata, and a reference to a brand. When pages are published, they become available at their configured URL path. The visual editor supports both desktop and mobile preview modes.

Brands store the complete design token set for each visual identity. This includes colour scales, semantic colour mappings, typography settings, spacing values, and asset URLs. Brands are created once and can be referenced by any number of pages.

Multi-Brand Theming System

The theming system enables multiple brands to share the same codebase while maintaining distinct visual identities. It operates through four layers:

The definition layer stores brand configurations as structured data objects containing all design tokens. Each brand defines a complete colour scale from light to dark shades, semantic colour mappings for primary actions, secondary elements, accents, backgrounds, and status indicators, plus typography and spacing values.

The resolution layer determines which brand applies to each page request. It first checks if the page content includes a brand reference. If found, it fetches that brand's configuration. As a fallback, it checks for a brand identifier in the URL query string. If neither exists, the default brand applies.

The injection layer converts the resolved brand configuration into CSS custom properties. These variables are injected as inline styles on a wrapper element, scoping them to the page content. This approach ensures brand styles don't leak between different brand pages and enables instant theming without stylesheet switching.

The consumption layer is where components read their styles. All brand-aware components use CSS variable references for colours and can access brand data directly via a React context hook for assets like logos.

Visitor Identification System

The visitor identification system provides persistent tracking of anonymous visitors across sessions. It uses a serverless architecture with a Cloud Function handling identification logic and Firestore providing persistent storage.

When a visitor first arrives, the browser SDK calls the identification API. The API generates a UUID, creates a visitor document in Firestore with initial metadata, and returns the ID to the browser. The SDK stores this ID in both a cookie and localStorage for redundancy.

On subsequent visits, the SDK retrieves the stored ID and sends it to the API. The API looks up the existing visitor document, increments the page view counter, updates the last seen timestamp, and returns the full profile including any accumulated behavioural data.

The visitor document stores identity information, engagement metrics, traffic source attribution, a chronological list of tracked events, computed audience segments, and a lead score. This data enables personalisation and provides analytics on visitor behaviour.

Lead Scoring

The system automatically computes a lead score based on visitor behaviour. Different events contribute different point values reflecting their indication of purchase intent. Basic engagement like page views contributes minimal points, while high-intent actions like form completions or contact requests contribute significantly more.

The lead score accumulates over time as visitors interact with the site. This score can be used for personalisation decisions, such as showing different content to high-intent versus casual visitors, or for prioritising leads in sales workflows.

Google Tag Manager Integration

The visitor SDK automatically pushes identification data to the GTM data layer when a visitor is identified. This includes the visitor ID, whether they're new or returning, their page view count, session count, lead score, and any assigned segments.

This integration enables GTM tags and triggers to access visitor data for analytics, advertising pixels, and other marketing tools without additional implementation work.

API Connections

Builder.io Content API

The application connects to Builder.io's Content Delivery Network for fetching pages and brand data. Content is retrieved using the official SDK which handles query parameter construction, caching headers, and visual editor integration automatically.

Page content is fetched by URL path, returning the full component tree and associated data. Brand content is fetched either by reference ID when linked from a page or by slug for direct lookups. Both endpoints support query-based filtering and pagination.

Content responses are cached using Incremental Static Regeneration with a revalidation period. This balances content freshness with API efficiency, ensuring published changes appear within the revalidation window while avoiding unnecessary API calls for unchanged content.

Builder.io Write API

For programmatic content management, the Write API enables creating and updating content entries. This is used by maintenance scripts for tasks like creating new brand entries or bulk-updating content fields. The Write API requires authentication with a private API key that is never exposed to the browser.

Visitor Identification API

The visitor identification API is a Cloud Function that handles both GET and POST requests. GET requests retrieve or create visitor profiles. POST requests track events and update visitor data.

The API accepts the visitor ID (if known), current page URL, referrer, and user agent. It returns the complete visitor profile including all stored data. For event tracking, it also accepts an event name and optional properties object.

Contact Request API

The application includes an API endpoint for processing contact form submissions. This endpoint validates the form data, constructs a contact request payload, and forwards it to the Access Group's contact management system. It handles error cases gracefully and returns appropriate status codes to the frontend.

Functionality

Equipment Enquiry System

The equipment enquiry system allows visitors to build a list of equipment they're interested in and submit a grouped enquiry. This workflow reduces friction compared to enquiring about each item individually.

The enquiry cart is managed through a React context that persists state to localStorage. This ensures the cart survives page navigation and browser refreshes. The context provides methods for adding items, removing items, checking if an item is in the cart, and clearing the cart entirely.

Equipment cards throughout the site display an "Add to Enquiry" button. When clicked, the item is added to the cart with its ID, name, brand, category, and image URL. Visual feedback shows when an item is already in the cart, and clicking again removes it.

A floating bubble appears in the corner when items are in the cart. It displays the current item count and animates when new items are added to draw attention. Clicking the bubble opens the enquiry panel.

The enquiry panel slides out to show all selected equipment with the ability to remove individual items. Below the equipment list, a form collects contact details including name, phone, email, company, industry, preferred branch, and project location. Submitting the form sends the enquiry to the contact API and clears the cart on success.

A quick view modal provides detailed equipment information without leaving the current page. It displays multiple images with thumbnail navigation, specifications, pricing tiers, and power/environment badges. The modal includes both an add-to-enquiry button and a direct call button.

Content Personalisation

The Hero component supports URL-based personalisation through campaign parameters. When visitors arrive via marketing campaigns, the URL typically includes tracking parameters identifying the campaign source and content variant.

The personalisation system reads these parameters and builds a context object containing the campaign identifier, any category targeting, and override content. This context can modify the headline, subheadline, background image, and call-to-action text.

Category-specific campaigns can highlight relevant keywords in the headline using the brand's primary colour. Geographic parameters can localise content to specific regions. The system falls back to default content when no personalisation parameters are present.

Visual Editing

Builder.io's visual editor is fully integrated into the application. When content editors access a page through the Builder.io interface, the application detects the editing context and adjusts its behaviour accordingly.

In editing mode, the page renders with additional metadata that enables the visual editor's drag-and-drop functionality. Editors can select any registered component, modify its properties through the sidebar panel, and see changes reflected immediately in the preview.

Components are registered with Builder.io including their property schemas, default values, and friendly names. This registration enables the editor to present appropriate input controls for each property type, from simple text fields to complex nested configurations.

Component Library

The application includes a comprehensive library of components registered for use in Builder.io:

Layout components include a brand-aware header with logo switching based on the active brand, and a footer with location-based contact information, social links, and brand-appropriate styling.

Hero components provide full-width introductory sections with configurable background images, overlay intensity, headlines with optional category highlighting, subheadlines, and primary/secondary call-to-action buttons. They support personalisation overrides and multiple height presets.

Content components include a rich text block with WYSIWYG editing capabilities and a contact form with full validation, API integration, and configurable field visibility.

Equipment components include a grid layout that fetches and displays equipment from the API with filtering capabilities, individual equipment cards with pricing and specifications, a search interface with category and brand filters, and a detail modal for quick viewing.

Landing page components provide a complete toolkit for building conversion-focused pages, including trust badges, benefits sections, testimonials, FAQ accordions, call-to-action banners, and sticky forms.

All components use CSS variables for theming and automatically adapt to the active brand without any component-level configuration.

CI/CD Process

Source Control

The codebase is managed in a Git repository with the main branch serving as the production branch. All changes flow through the main branch, with feature branches used for development work that requires isolation.

Automatic Deployment

The hosting platform monitors the repository for changes. When commits are pushed to the main branch, an automatic deployment pipeline triggers. The pipeline executes the following sequence:

First, the latest code is pulled from the repository. Dependencies are installed from the package lock file to ensure reproducible builds. The build process then compiles TypeScript, bundles the application, and generates optimised production assets.

During the build, static pages are pre-rendered where possible, and serverless functions are created for dynamic routes. The build output is then deployed to the edge network, replacing the previous deployment atomically to prevent any downtime.

Finally, the CDN cache is invalidated for changed assets, and the new deployment goes live. The entire process typically completes within one to two minutes of the triggering commit.

Preview Deployments

For pull requests, the platform automatically creates preview deployments on unique URLs. This allows changes to be tested in a production-like environment before merging. Preview URLs are posted as comments on the pull request for easy access.

Manual Deployment

When immediate deployment is needed or cache invalidation is required, manual deployment can be triggered through the platform's CLI. A force flag bypasses build caching to ensure completely fresh builds when troubleshooting caching issues.

Cloud Function Deployment

The visitor identification Cloud Function is deployed separately through Google Cloud Platform's tooling. Deployment can be triggered via npm script or directly through the gcloud CLI. The function automatically scales based on traffic and requires no capacity planning.

Brand Configuration

Default Brand

The default brand uses a red colour scheme reflecting Access Hire Australia's established identity. The header uses a white background with dark text for contrast, while the footer uses the brand red with white text. Typography pairs a geometric sans-serif for headings with a humanist sans-serif for body text.

Secondary Brand

The secondary brand uses a navy colour scheme for a more corporate appearance. Both header and footer use the deep navy background with white text, creating a distinctive look that immediately differentiates it from the primary brand. An orange secondary colour provides vibrant call-to-action buttons, while a blue accent colour handles links and highlights. Typography uses different font families to further distinguish the brand identity.

Adding New Brands

New brands can be added through the CMS interface by creating a new brand entry and populating all required colour, typography, and asset fields. Alternatively, brands can be added to the local configuration file for developer-managed brands. Once created, brands become immediately available for selection when editing pages.

Performance Considerations

Server-side rendering ensures fast initial page loads by delivering fully-rendered HTML before JavaScript executes. This also ensures content is indexable by search engines without requiring JavaScript execution.

CSS variable theming operates entirely in CSS, avoiding JavaScript overhead for style calculations. Theme switching between brands requires no additional network requests or style recalculation.

Content caching through Incremental Static Regeneration reduces API calls to the CMS while ensuring published changes propagate within a reasonable timeframe. Static assets are served from edge locations geographically close to visitors.

Image optimisation happens automatically through the framework's image component, which handles responsive sizing, format selection, and lazy loading without manual intervention.

Font loading uses the display swap strategy to prevent invisible text during font loading, with fallback system fonts specified to maintain layout stability.

Security Considerations

API keys are segregated by capability. Public keys used for content fetching are safe to expose in client-side code as they only permit read operations. Private keys for write operations are stored as environment variables and never included in client bundles.

The contact form API validates all input server-side before forwarding to downstream systems. This prevents malformed data from reaching the contact management system and protects against injection attacks.

Visitor tracking stores only anonymous behavioural data. No personally identifiable information is collected through the tracking system. The visitor ID is a randomly generated UUID with no correlation to real-world identity.

All external communications use HTTPS encryption. The application enforces secure connections and sets appropriate security headers through the hosting platform's configuration.